Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Ciência da Computação

Parsing Expression Grammar with Syntactic Attributes

Uma Formalização em PLT Redex

Gabriel Pires Ferreira

JUIZ DE FORA
SETEMBRO, 2024

Parsing Expression Grammar with Syntactic Attributes

Uma Formalização em PLT Redex

Gabriel Pires Ferreira

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientador: Leonardo do Santos Vieira Reis

Coorientador: Elton Máximo Cardoso

JUIZ DE FORA

SETEMBRO, 2024

PARSING EXPRESSION GRAMMAR WITH SYNTACTIC ATTRIBUTES

Uma Formalização em PLT Redex

Gabriel Pires Ferreira

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

Leonardo do Santos Vieira Reis
Doutor em Ciência da Computação/UFMG

Elton Máximo Cardoso
Mestre em Ciência da Computação/UFOP

Rodrigo Geraldo Ribeiro
Doutor em Ciência da Computação/UFMG

Gleiph Ghiotto Lima de Menezes
Doutor em Ciência da Computação/UFF

JUIZ DE FORA

2 DE SETEMBRO, 2024

Resumo

Neste trabalho, descrevemos uma formalização de *Parsing Expression Grammar with Syntactic Attributes* (PEGwSA), formalismo que descreve analisadores sintáticos *top-down*. PEGwSA é uma extensão de *Parsing Expression Grammar* (PEG) que acrescenta atributos e operadores para manipulá-los. Além disso, PEGwSA é a base para *Adaptable Parsing Expression Grammar* (APEG), formalismo que introduz mecanismos que permitem a manipulação dinâmica das regras que compõem a gramática.

Especificamos em PLT Redex um sistema de tipos e semânticas operacionais *small-step* e *big-step* para PEGwSA. Reunidas, essas especificações constituem a formalização de PEGwSA apresentada neste trabalho.

PLT Redex é uma linguagem de domínio específico projetada para formalizar modelos semânticos que oferece uma vasta gama de ferramentas para desenvolver e aplicar testes a esses modelos. Com o auxílio dessas ferramentas, testamos a formalização de PEGwSA, processo que levou à descoberta de erros e inconsistências, principalmente em relação ao sistema de tipos.

Em virtude dos erros e inconsistências descobertos, desenvolvemos um sistema de tipos para PEGwSA que agrega ao sistema de tipos para APEG apresentado por Cardoso et al. (2019), tendo em vista sua capacidade de capturar erros que seu predecessor ignora.

Palavras-chave: PEG, PEGwSA, parsing expression grammar, atributo, semântica, sistema de tipos.

Abstract

In this work, we describe a formalization of PEGwSA, a formalism that describes top-down parsers. PEGwSA is an extension of PEG that adds attributes and operators to manipulate them. Furthermore, PEGwSA serves as the basis for APEG, a formalism that introduces mechanisms for the dynamic manipulation of the rules that make up the grammar.

We specify in PLT Redex a type system and both small-step and big-step operational semantics for PEGwSA. Together, these specifications constitute the formalization of PEGwSA presented in this work.

PLT Redex is a domain-specific language designed to formalize semantic models, providing a wide range of tools for developing and testing these models. With the help of these tools, we tested the PEGwSA formalization, a process that led to the discovery of errors and inconsistencies, mainly related to the type system.

Due to the discovered errors and inconsistencies, we developed a type system for PEGwSA that improves upon the type system for APEG presented by Cardoso et al. (2019), given its ability to capture errors that its predecessor overlooks.

Keywords: PEG, PEGwSA, parsing expression grammar, attribute, semantics, type system.

Agradecimentos

À minha família, pelo encorajamento e apoio incondicional.

Ao corpo docente da UFJF, pelos serviços prestados ao longo de todos esses anos de curso.

Aos professores Leonardo Vieira dos Santos Reis, Elton Máximo Cardoso e Rodrigo Geraldo Ribeiro, pela disponibilidade em esclarecer dúvidas e solucionar problemas, muitas vezes surgidos na última hora, ao longo do desenvolvimento desse trabalho, assim como pelos valiosos conselhos e palavras de incentivo.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

APEG	*Adaptable Parsing Expression Grammar*
DCC	Departamento de Ciência da Computação
ICE	Instituto de Ciências Exatas
PEG	*Parsing Expression Grammar*
PEGwSA	*Parsing Expression Grammar with Syntactic Attributes*
UFJF	Universidade Federal de Juiz de Fora

1 Introdução

Modelos formais desempenham diversas funções na comunidade de linguagens de programação (KLEIN et al., 2012). Destacamos, dentre essas funções, a de definir as regras sintáticas e semânticas, assim como o sistema de tipos, que, uma vez reunidos, descrevem uma linguagem de programação.

Pesquisadores de linguagens de programação utilizam modelos formais para comunicar ideias de maneira concisa e precisa. Por décadas, eles estiveram utilizando *paper-and-pencil* para formalizar estes modelos, mas, devido a informalidade deste método, modelos especificados em *paper-and-pencil* costumam apresentar falhas. Estas falhas levam a ruídos na comunicação e, algumas vezes, sobrevivem aos processos de revisão destes modelos (KLEIN et al., 2012).

Independentemente da função que desempenha, um modelo defeituoso não serve ao seu propósito. Uma maneira de eliminar falhas de um modelo é especificá-lo em uma linguagem formal mecanizada, tal como PLT Redex. PLT Redex é uma linguagem mecanizada de domínio específico, embutida na linguagem de programação Racket, voltada para a formalização de modelos semânticos. PLT Redex oferece uma série de ferramentas para desenvolver e aplicar testes sobre esses modelos (KLEIN et al., 2012).

Neste trabalho objetivamos formalizar, utilizando a linguagem PLT Redex, *Parsing Expression Grammar with Syntactic Attributes* (PEGwSA), um formalismo para descrever analisadores sintáticos *top-down* que estende *Parsing Expression Grammar* (PEG) (FORD, 2004) com atributos sintáticos e operadores para manipulá-los. PEGwSA é a base para *Adaptable Parsing Expression Grammar* (APEG) (CARDOSO et al., 2019), formalismo que estende PEGwSA com mecanismos que permitem a manipulação dinâmica das regras de produção que constituem a gramática.

Formalizamos PEGwSA em PLT Redex com o objetivo de encontrar e corrigir falhas no modelo. Para tal, especificamos um sistema de tipos e semânticas operacionais *big-step* e *small-step* para PEGwSA. Com o auxílio das ferramentas disponibilizadas pela linguagem PLT Redex, desenvolvemos e aplicamos uma série de testes sobre essas

especificações, processo que levou à descoberta de falhas no modelo, principalmente falhas relacionadas ao sistema de tipos. A formalização de PEGwSA em PLT Redex pode ser encontrada em <https://github.com/lives-group/apeg-redex/tree/PEGwSA>.

Ao todo, encontramos quatro falhas relacionadas às expressões de atributos em algumas das regras de tipagem que tratam expressões de *parsing*, mais precisamente as regras que atuam sobre escolhas ordenadas, negações, repetições e *binds*. Essas regras de tipagem permitiam referências a atributos não-instanciadas ou cujos tipos são indefinidos, além de equivocadamente introduzir polimorfismo no sistema de tipos.

Graças às falhas descobertas, aperfeiçoamos a formalização de PEGwSA, desenvolvendo um sistema de tipos que captura erros ignorados por seus predecessores. As adaptações que aperfeiçoam o sistema de tipos para PEGwSA que desenvolvemos podem ser aplicadas ao sistema de tipos para APEG proposto por Cardoso et al. (2019), estabelecendo uma importante contribuição para esse trabalho. Em suma, as contribuições deste trabalho são:

- definição de uma semântica *small-step* para PEGwSA;
- formalização da semântica *small-step* e *big-step* de PEGwSA em PLT Redex;
- identificação de falhas no sistema de tipos de Cardoso et al. (2019);
- especificação de um sistema de tipos para PEGwSA;
- formalização do sistema de tipos proposto em PLT Redex;
- execução de testes com as especificações em PLT Redex.

O restante deste documento está organizado da seguinte forma: no Capítulo 2 apresentamos os trabalhos relacionados ao nosso. Em seguida, no Capítulo 3 apresentamos PLT Redex e Parsing Expression Grammars, conceitos fundamentais para o entendimento do presente trabalho. No Capítulo 4 é apresentado a formalização de PEGwSA e, por fim, o Capítulo 5 conclui o trabalho.

2 Trabalhos Relacionados

Preocupados com a escassez de ferramentas eficazes para identificar erros em programas escritos na linguagem de programação Python, com a crescente popularidade dessa linguagem e com sua adoção em domínios cada vez mais importantes, Politz et al. (2013) apresentaram uma semântica operacional *small-step* para a linguagem de programação Python e desenvolveram uma linguagem núcleo para Python, além de um processo para converter código escrito em Python para essa linguagem núcleo. A semântica desenvolvida por eles, que é traçável por ferramentas e provas, auxilia a suprir a crescente demanda por uma semântica precisa para analisar programas escritos em Python e provar propriedades sobre eles.

Em um desdobramento semelhante, Guha, Saftoiu e Krishnamurthi (2010) — confrontados pela ascendente popularidade da linguagem de programação JavaScript, cujas numerosas peculiaridades muitas vezes são exploradas por agentes maliciosos em violações de segurança e privacidade — formalizam uma linguagem núcleo para JavaScript, assim como uma semântica operacional *small-step* para essa linguagem núcleo, por meio da qual construíram um subconjunto seguro de JavaScript.

Por fim, análoga a Python e JavaScript, a linguagem de programação PHP, uma das linguagens mais comumente usadas para *scripting* do lado servidor na comunicação cliente-servidor, possui diversas características dinâmicas que facilitam a introdução de erros em programas, o que abre brechas para o vazamento de informações sensíveis e outras formas de comprometimento, aspecto ainda mais preocupante tendo em vista o uso da linguagem em sistemas complexos, tais como bancos online, redes sociais e computação em nuvem (FILARETTI; MAFFEIS, 2014). À vista disso, Filaretti e Maffeis (2014) apresentam a primeira semântica formal executável para PHP, que pode servir como base para definir ferramentas de verificação baseadas em semântica.

Estes trabalhos elucidam os benefícios oriundos da formalização de linguagens de programação e servem como guias para formalizar outros modelos semânticos, especificamente *Parsing Expression Grammar with Syntactic Attributes* (PEGwSA).

3 Fundamentos Teóricos

3.1 Semântica Operacional e Sistema de Tipos

Uma semântica operacional (PIERCE, 2002) especifica o comportamento de uma linguagem ao definir uma máquina abstrata para processar os termos que a compõem. Pierce (2002) destaca dois estilos de semântica operacional: *small-step* e *big-step*. Em uma semântica operacional *small-step*, passos individuais de computação são usados para reescrever um termo, pouco a pouco, até que eventualmente ele se torne um valor. Uma semântica operacional *big-step*, por outro lado, emprega múltiplos passos de uma só vez, levando um termo diretamente ao valor correspondente.

Um sistema de tipos (PIERCE, 2002) é um método sintático que verifica automaticamente a ausência de determinados comportamentos errôneos em um programa, classificando os termos que o compõem de acordo com os valores para os quais eles são avaliados.

Neste trabalho, descrevemos um sistema de tipos e semânticas operacionais *big-step* e *small-step* para PEGwSA.

3.2 PLT Redex

PLT *Redex* (KLEIN et al., 2012) é uma linguagem executável de domínio específico embutida na linguagem de programação *Racket* projetada para mecanização de modelos semânticos. Usando PLT *Redex*, engenheiros semânticos descrevem gramáticas, reduções e meta-funções de semânticas operacionais e sistema de tipos. PLT *Redex* oferece um vasto conjunto de ferramentas que cobrem uma variedade de tarefas relacionadas à execução de definições semânticas: um *stepper* para semântica *small-step*, inspetores para gráficos de reduções, um *framework* de testes unitários, uma ferramenta de testes automatizados, etc.

A fim de introduzir PLT Redex, pegamos emprestada uma simples linguagem

```
1 (define-language Pierce
2   [t ::= true false 0
3     (if t t t)
4     (succ t)
5     (pred t)
6     (iszero t)])
```

Figura 3.1: Especificação em PLT *Redex* da linguagem *Pierce*.

tipada de Pierce (2002), cuja a sintaxe é detalhada na Figura 3.1.

PLT Redex provê a função `define-language` para modelar linguagens, essa função define uma gramática livre de contexto que descreve a sintaxe da linguagem modelada. Ao ser chamada, os seguintes parâmetros devem ser fornecidos à função `define-language`: o nome e as definições dos não-terminais que compõem a linguagem. No exemplo em questão (Figura 3.1), a linguagem, que nomeamos como `Pierce`, é composta por um único não-terminal, `t`, que pode ser um booleano — `true` ou `false` —, um número natural — `0`, `(succ t)` ou `(pred t)` —, uma expressão condicional — `(if t t t)` — ou uma expressão relacional — `(iszero t)`.

```
1 (reduction-relation Pierce
2   (--> (if true t_1 t_2) t_1 "if-true")
3   (--> (if false t_1 t_2) t_2 "if-false")
4   (--> (iszero 0) true "=0")
5   (--> (iszero (succ t)) false "/=0")
6   (--> (pred 0) 0 "pred0")
7   (--> (pred (succ t)) t "pred-succ"))
```

Figura 3.2: Relação de redução da linguagem Pierce.

Com o intuito de definir a semântica da linguagem Pierce, empregamos a função `reduction-relation` para descrever o conjunto de regras de reescrita, que definem como um termo deve ser reescrito. Uma regra de reescrita tem a forma `(-> <termo> <termo reescrito> <nome da regra>)`, que determina que o `<termo reescrito>` deve substituir o `<termo>`. O parâmetro facultativo `<nome da regra>` é uma cadeia de caracteres que nomeia a regra de reescrita. As regras `"if-true"` e `"if-else"` definem, respectivamente, que uma expressão condicional `(if t t_1 t_2)` deve ser reduzida para `t_1` se `t` é `true` e para `t_2` se `t` é `false`. As regras `"=0"` e `"/=0"` determinam, respectivamente, que a operação `(iszero t)` reduz para *true* se t é `0` e para *false* se t é `(succ t_1)`, ou seja, se `t` é o

sucessor de um número natural. Por fim, as regras `"pred0"` e `"pred-succ"` definem que o predecessor de um número natural `(pred t)` reduz para `0` se `t` é `0` e para `t_1` se `t` é `(succ t_1)`.

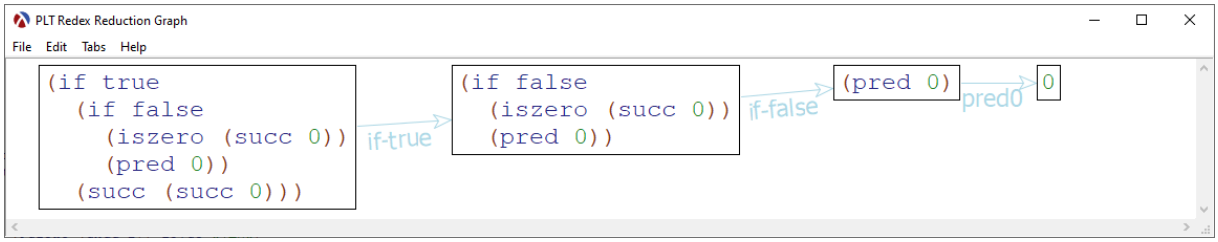

Figura 3.3: Uma captura de tela do visualizador de redução de Redex.

Na Figura 3.3 apresentamos uma captura de tela do visualizador de redução de PLT Redex do termo `(if true (if false (iszero (succ 0) (pred 0)) (succ (succ 0))))`.

Termos da linguagem Pierce podem ser assinalados com um dentre dois tipos: booleano (`Bool`) e natural (`Nat`). Utilizamos a função `define-judgment-form` para modelar o sistema de tipos da linguagem Pierce, que associa a cada termo `t` o seu tipo apropriado `T`.

As regras de tipagem que constituem o sistema de tipos são apresentadas como regras de inferência (Figura 3.4), em que a `<conclusão>` só pode ser alcançada se as `<premissas>` são verdadeiras. O `<nome da regra>` é um parâmetro opcional que rotula a regra que integra.

A Figura 3.5 retrata a especificação do sistema de tipos para a linguagem Pierce. As regras `"true"` e `"false"` definem que os literais booleanos `true` e `false` são assinalados com o tipo `Bool`. A regra `"if"` define que uma expressão condicional `(if t_1 t_2 t_3)` é assinalada com o mesmo tipo para o qual ambas suas subexpressões `t_2` e `t_3` são avaliadas, assim como presume que a subexpressão teste, `t_1`, é avaliada para `Bool`. A regra `"0"` define que o termo `0` é assinalado com o tipo `Nat`. As regras `"succ"` e `"pred"` definem que sucessores `(succ t)` e predecessores `(pred t)` de números

```
1  [<premissas>
2  ----------- <nome da regra>
3  <conclusão>]
```

Figura 3.4: Estrutura da regra de inferência.

```
(define-judgment-form TypePierce
  #:mode (: I O)
  #:contract (: t T)

  [------------- "true"
   (: true Bool)]

  [-------------- "false"
   (: false Bool)]

  [(: t_1 Bool)
   (: t_2 T)
   (: t_3 T)
   ---------------------- "if"
   (: (if t_1 t_2 t_3) T)]

  [--------- "0"
   (: 0 Nat)]

  [(: t Nat)
   ---------------- "succ"
   (: (succ t) Nat)]

  [(: t Nat)
   ---------------- "pred"
   (: (pred t) Nat)]

  [(: t Nat)
   ------------------- "iszero"
   (: (iszero t) Bool)])
```

Figura 3.5: Sistema de tipos da linguagem Pierce.

naturais são assinalados com o tipo `Nat` se `t` também é avaliado para `Nat`. Por fim, a regra `"iszero"` define que a operação `(iszero t)` é assinalada com `Bool` se `t` é avaliado para `Nat`.

3.3 Parsing Expression Grammar

PEG (FORD, 2004) é um formalismo baseado em reconhecimento projetado para descrever analisadores sintáticos *top-down*. Formalmente, uma PEG (CARDOSO et al., 2023) é uma quadrupla $G = (V, \Sigma, R, p_{inicial})$, em que V é um conjunto finito que abriga os símbolos não-terminais, Σ é o alfabeto, R é o conjunto finito de regras de produção e $p_{inicial}$, a expressão de *parsing* inicial. Uma regra de produção $N \leftarrow p$ é uma associação

entre um não-terminal N e uma expressão p.

$$
\begin{array}{l}
p ::= t \mid \varepsilon \mid p \bullet p \mid p/p \mid p* \mid !p \mid N \\
\phi ::= t_1 ... \, t_n \\
w ::= (\phi) \\
o ::= w \mid \bot
\end{array}
$$

Figura 3.6: Sintaxe abstrata de expressão de *parsing*, entrada e resultado.

A Figura 3.6 apresenta as sintaxes de expressão de *parsing*, entrada e resultado. Uma expressão de *parsing* pode ser um terminal (t), uma cadeia vazia (ε), uma sequência ($p \bullet p$), uma escolha ordenada (p/p), uma repetição $p*$, uma negação $!n$ ou uma chamada de não-terminal N. Os símbolos ϕ e w retratam sequências de terminais, esse último exige que a sequência representada esteja limitada por parenteses. O símbolo o denota o resultado da aplicação de uma expressão de *parsing* a uma entrada, que pode ser sucesso (w) ou falha ($\bot$). O símbolo w ora representa a entrada, ora representa o prefixo da entrada consumido por uma expressão.

A Figura 3.7 apresenta as regras que constituem a semântica *big-step* de PEG. O julgamento $G \vdash p, w \Rightarrow o$ define que a aplicação da expressão de *parsing* p à entrada w resulta em o. Nas vezes em que a aplicação de uma expressão de *parsing* resulta em sucesso, o resultado $o = w'$ abriga o prefixo da entrada consumido pela expressão.

A execução de um símbolo terminal é bem-sucedida, consumindo-o, se a entrada é iniciada com o mesmo símbolo (regra Terminal) e falha se a entrada é ou iniciada por um símbolo diferente (regra $\neg$Terminal$_1$) ou vazia (regra $\neg$Terminal$_2$). A execução de uma cadeia vazia sempre resulta em sucesso sem consumir a entrada (regra Cadeia Vazia).

A execução de uma sequência resulta em sucesso se a aplicação sucessiva de suas subexpressões for bem-sucedida (regra Sequência). As regras $\neg$Sequência$_1$ e $\neg$Sequência$_2$ definem que uma sequência falha se a execução de alguma de suas subexpressões falhar. A execução de uma alternativa é bem-sucedida se a aplicação de alguma de suas subexpressões resultar em sucesso. As regras que tratam escolhas ordenadas definem que a segunda alternativa só deve ser aplicada se a primeira falhar.

Aplicar uma repetição $p*$ a uma entrada equivale a repetidamente aplicar sua

$$\boxed{G \vdash p, w \Rightarrow o}$$

$$\frac{}{G \vdash t, (t\ \phi) \Rightarrow (t)}\ \textsc{Terminal} \qquad \frac{t \neq t'}{G \vdash t, (t'\ \phi) \Rightarrow \bot}\ \neg\textsc{Terminal}_1$$

$$\frac{}{G \vdash t, () \Rightarrow \bot}\ \neg\textsc{Terminal}_2 \qquad \frac{}{G \vdash \varepsilon, w \Rightarrow ()}\ \textsc{Cadeia Vazia}$$

$$\frac{\begin{array}{c} G \vdash p_1, (\phi_1\ \phi_2\ \phi_3) \Rightarrow (\phi_1) \\ G \vdash p_2, (\phi_2\ \phi_3) \Rightarrow (\phi_2) \end{array}}{G \vdash p_1 \bullet p_2, (\phi_1\ \phi_2\ \phi_3) \Rightarrow (\phi_1\ \phi_2)}\ \textsc{Sequência} \qquad \frac{G \vdash p_1, w \Rightarrow \bot}{G \vdash p_1 \bullet p_2, w \Rightarrow \bot}\ \neg\textsc{Sequência}_1$$

$$\frac{\begin{array}{c} G \vdash p_1, (\phi_1\ \phi_2) \Rightarrow (\phi_1) \\ G \vdash p_2, (\phi_2) \Rightarrow \bot \end{array}}{G \vdash p_1 \bullet p_2, (\phi_1\ \phi_2) \Rightarrow \bot}\ \neg\textsc{Sequência}_2 \qquad \frac{G \vdash p_1, (\phi_1\ \phi_2) \Rightarrow (\phi_1)}{G \vdash p_1/p_2, (\phi_1\ \phi_2) \Rightarrow (\phi_1)}\ \textsc{Escolha Ordenada}_1$$

$$\frac{\begin{array}{c} G \vdash p_1, (\phi_1\ \phi_2) \Rightarrow \bot \\ G \vdash p_2, (\phi_1\ \phi_2) \Rightarrow (\phi_1) \end{array}}{G \vdash p_1/p_2, (\phi_1\ \phi_2) \Rightarrow (\phi_1)}\ \textsc{Escolha Ordenada}_2$$

$$\frac{\begin{array}{c} G \vdash p_1, w \Rightarrow \bot \\ G \vdash p_2, w \Rightarrow \bot \end{array}}{G \vdash p_1/p_2, w \Rightarrow \bot}\ \neg\textsc{Escolha Ordenada} \qquad \frac{\begin{array}{c} G \vdash p, (\phi_1\ \phi_2\ \phi_3) \Rightarrow (\phi_1) \\ G \vdash p*, (\phi_2\ \phi_3) \Rightarrow (\phi_2) \end{array}}{G \vdash p*, (\phi_1\ \phi_2\ \phi_3) \Rightarrow (\phi_1\ \phi_2)}\ \textsc{Repetição}_1$$

$$\frac{G \vdash p, w \Rightarrow \bot}{G \vdash p*, w \Rightarrow ()}\ \textsc{Repetição}_2 \qquad \frac{G \vdash p, w \Rightarrow \bot}{G \vdash !p, w \Rightarrow ()}\ \textsc{Negação}$$

$$\frac{G \vdash p, (\phi_1\ \phi_2) \Rightarrow (\phi_1)}{G \vdash !p, (\phi_1\ \phi_2) \Rightarrow \bot}\ \neg\textsc{Negação} \qquad \frac{\begin{array}{c} G[\![N]\!] = p \\ G \vdash p, (\phi_1\ \phi_2) \Rightarrow (\phi_1) \end{array}}{G \vdash N, (\phi_1\ \phi_2) \Rightarrow (\phi_1)}\ \textsc{Não-terminal}$$

$$\frac{\begin{array}{c} G[\![N]\!] = p \\ G \vdash p, w \Rightarrow \bot \end{array}}{G \vdash N, w \Rightarrow \bot}\ \neg\textsc{Não-terminal}$$

Figura 3.7: Semântica *big-step* de PEG.

subexpressão p à entrada. Quando sua subexpressão falha, a repetição resulta em sucesso sem consumir a entrada. A execução de uma negação falha se sua subexpressão for bem-sucedida (regra Negação) e resulta em sucesso sem consumir a entrada se a aplicação de sua subexpressão falhar (regra ¬Negação).

Por fim, a execução de uma chamada de não-terminal é bem-sucedida se a aplicação de sua expressão de *parsing* associada resultar em sucesso (regra Não-terminal)

e falha se a aplicação dessa expressão falhar (regra ¬NÃO-TERMINAL). A notação $G[\![N]\!]$ denota a expressão de *parsing* associada ao não-terminal N na gramática G.

4 Parsing Expression Grammar with Syntactic Attributes

Neste capítulo apresentamos a formalização em PLT Redex de PEGwSA. Esta extensão de PEG é a base para o modelo APEG, que usa o conceito de atributos para formalizar um mecanismo de adaptabilidade dinâmica do conjunto de regras da gramática. Para um melhor entendimento da formalização, dividimos a apresentação do modelo em: sintaxe (Seção 4.1), valores (Seção 4.2), semântica *big-step* (Seção 4.3), semântica *small-step* (Seção 4.4) e sistema de tipos (Seção 4.5).

4.1 Sintaxe de PEGwSA

$$
\begin{array}{rcl}
\tau & ::= & \mathit{Bool} \mid \mathit{Integer} \mid \mathit{String} \mid < \tau > \mid [\tau] \mid \overline{\tau} \rightarrow \overline{\tau} \\
e & ::= & \mathbf{true} \mid \mathbf{false} \mid i \mid s \mid < \overline{\overline{e/e}} > \mid e : e \mid \mathbf{nil} \mid e + e \\
 & & \mid e - e \mid e \times e \mid e \div e \mid e == e \mid e > e \mid e \wedge e \mid e \vee e \\
 & & \mid \neg e \mid \vartheta \mid \mathbf{get}\ e\ e \mid \mathbf{put}\ e\ e\ e \mid \mathbf{head}\ e \mid \mathbf{tail}\ e \\
p & ::= & p \bullet p \mid p/p \mid !p \mid p* \mid N\ \overline{e}\ \overline{\vartheta} \mid \vartheta = p \mid \vartheta \leftarrow e \\
 & & \mid ?e \mid t \mid \varepsilon \\
G & ::= & \overline{N_! / \overline{\vartheta\ ::\ \tau}\ \overline{e} \rightarrow p}
\end{array}
$$

Figura 4.1: Sintaxe abstrata de PEGwSA.

A Figura 4.1 apresenta a sintaxe abstrata de PEGwSA. Para facilitar a compreensão, a sintaxe foi dividida em expressões de atributos (e) e expressões de *parsing* (p).

É preciso se atentar a três notações empregadas nessa definição de sintaxe: $\overline{\overline{x}}$, $\overline{x}$ e $x_!$. As notações $\overline{\overline{x}}$ e $\overline{x}$ são utilizadas para denotar sequências de zero ou mais termos, essa última requer que a sequência representada esteja limitada por parênteses, e.g. $(\overline{\overline{x}}) = \overline{x} = (x_1\ x_2\ x_3 ...\ x_n)$. Por vezes, utilizamos essas notações com o comprimento explicitamente grafado, e.g. $(\overline{\overline{x}}^n) = \overline{x}^n = (x_1 x_2 x_3 ... x_n)$. A notação $x_!$ demanda que o termo encapsulado, x, seja único, e.g. $\overline{x_!}^n = (x_1\ x_2\ x_3 ...\ x_n)$ tal que $\forall i, j \in \{1, 2, 3, ..., n\} : x_i = x_j \implies i = j$.

```
(define-language AttributeL
  [e e1 e2 e3 e4 ::= b
     i
     s
     (⇒ (e e) ...)
     (get e e)
     (put e e e)
     (: e e)
     nil
     (head e)
     (tail e)
     (+ e e)
     (× e e)
     (÷ e e)
     (- e e)
     (∧ e e)
     (∨ e e)
     (¬ e)
     (== e e)
     (> e e)
     ϑ]
  [b ::= boolean]
  [i i1 i2 ::= integer]
  [s s1 s2 ::= string]
  [ϑ ϑ1 ϑ2 ϑ3 ϑ4 ::= variable-not-otherwise-mentioned])
```

Figura 4.2: Especificação em PLT Redex da linguagem *AttributeL*, que define a sintaxe de expressão de atributos.

```
(define-extended-language AttributeLType AttributeL
  [τ τ1 τ2 τ3 τ4 ::= Bool
     Integer
     String
     (: τ)
     (⇒ τ)
     (→ (τ ...) (τ ...))]
  [γ ::= variable-not-otherwise-mentioned]
  [Γ Γ1 Γ2 ::= ((γ_!_ τ)...)])
```

Figura 4.3: Especificação em PLT Redex da linguagem *AttributeLType*, que define as sintaxes de tipo e ambiente de tipos.

A primeira regra, τ, retrata os tipos de expressões de atributos, e. Um tipo (τ) pode retratar booleanos (*Bool*), inteiros (*Integer*), cadeia de caracteres (*String*), mapas ($< \tau >$), listas ($[\tau]$) e, por fim, não-terminais ($\overline{\tau} \rightarrow \overline{\tau}$).

Expressões de atributos, e, podem ser literais; construtores; operações aritméticas, relacionais e lógicas; atributos (ϑ) e manipuladores de listas (**head** e e **tail** e) e mapas (**get** e e e **put** e e e). Um literal pode ser um booleano (**true** ou **false**), um inteiro (i) ou

```
(define-extended-language AttributePeg vAttributeL
  [p p1 p2 p3 ::= (NT [e ...] [ϑ ...])
     ([← ϑ e] ...)
     (= ϑ p)
     (? e)
     (• p p)
     (/ p p)
     (* p)
     (! p)
     t
     ε]
  [t t1 t2 t3 n n1 n2 n3 m ::= natural]
  [NT ::= variable-not-otherwise-mentioned]
  [G ::= ((NT_!_ [(τ ϑ) ...] [e ...] p) ...)]
  [r r1 r2 ::= (NT [(τ ϑ) ...] [e ...] p)])
```

Figura 4.4: Especificação em PLT Redex da linguagem *AttributePeg*, que define as sintaxes de expressão de *parsing* e gramática.

uma cadeia de caracteres (s). Três construtores foram definidos: um para tratar mapas ($< \overline{\overline{e/e}} >$), outro para construir listas ($e : e$) e um último para expressar listas vazias (nil).

Nessa formalização de PEGwSA, foram incluídas quatro operações aritméticas — adição ($e + e$), subtração ($e - e$), multiplicação ($e \times e$) e divisão ($e \div e$) — duas operações relacionais — igualdade ($e == e$) e maior que ($e > e$) — e três operações lógicas — conjunção ($e \wedge e$), disjunção ($e \vee e$) e negação lógica ($\neg e$).

Uma expressão de *parsing*, p, pode ser uma sequência ($p \bullet p$), uma escolha ordenada (p/p), uma negação ($!p$), uma repetição ($p*$), uma chamada de não-terminal ($N\ \overline{e}\ \overline{\vartheta}$), um *bind* ($\vartheta = p$), um *update* ($\vartheta \leftarrow e$), uma restrição ($?e$), um terminal ($t$) ou uma cadeia vazia ($\varepsilon$).

A última regra, G, define que uma PEGwSA é uma sequência finita de produções. Uma produção, por sua vez, é uma associação entre um não-terminal, N, e uma tripla formada por uma sequência de atributos herdados ladeados pelos seus respectivos tipos ($\overline{\vartheta :: \tau}$), uma sequência de expressões de atributos que serão posteriormente amarradas aos seus atributos sintetizados ($\overline{e}$) e uma expressão de *parsing* (p), que nada mais é que o corpo da regra de produção.

As Figuras 4.2, 4.3 e 4.4 apresentam, respectivamente, as especificações em PLT Redex das sintaxes de expressão de atributos, tipo e expressão de *parsing*. É importante

ressaltar que, em PLT Redex, ao invés de definir um termo para representar um único *update* $(\vartheta \leftarrow e)$, empregamos um açúcar sintático, `((← ϑ e) ...)`, que retrata uma sequência de zero ou mais *updates*.

$$
\begin{array}{l}
(S/()\ (value) \rightarrow T(0)\ (value) \\
\ T/(value :: Integer)\ (value) \rightarrow\ \text{`1'} \bullet T\ (2 \times value + 1)\ (value) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad / \text{`0'} \bullet T\ (2 \times value)\ (value) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad / \varepsilon)
\end{array}
$$

Figura 4.5: Exemplo de PEGwSA expressado através da sintaxe abstrata.

```
([S () (value) (T (0) (value))]
 [T ((Integer value)) (value)
    (/ (• 1 (T ((+ (× 2 value) 1)) (value)))
       (/ (• 0 (T ((× 2 value)) (value)))
          ε))])
```

Figura 4.6: Exemplo de PEGwSA expressado através da especificação da sintaxe em Redex.

As Figuras 4.5 e 4.6 retratam uma mesma PEGwSA, a primeira descrita através da sintaxe abstrata, e a última, da especificação dessa sintaxe em Redex. Esta PEGwSA processa um número natural consumindo sua respectiva representação no sistema de numeração binário.

4.2 Valor e ambiente

$$
\begin{array}{l}
v\ ::=\ \textbf{true} \mid \textbf{false} \mid i \mid s \mid\ < \overline{\overline{s_! / v}} >\ \mid v : v \mid \textbf{nil} \\
\Delta\ ::=\ \overline{\vartheta_! / v}
\end{array}
$$

Figura 4.7: Sintaxe abstrata de valor.

A Figura 4.7 apresenta a sintaxe de valor e ambiente. Um valor é um elemento de um subconjunto de expressões bem-tipadas que engloba todos os possíveis resultados finais das avaliações de expressões de atributos. Todo valor de expressão de atributos é literal. Um valor pode ser um booleano (**true** ou **false**), um inteiro (i), uma cadeia de caracteres (s), um mapa ($< \overline{\overline{s_! / v}} >$), uma lista ($v : v$) e, mais especificamente, uma

lista vazia (**nil**). É importante ressaltar que um mapa é um valor se, e somente se, ele mapeia de cadeia de caracteres (s) para valores de um tipo arbitrário (v). De maneira semelhante, uma lista pode ser considerada um valor se, e somente se, ambas cabeça ($\mathbf{v} : v$) e cauda ($v : \mathbf{v}$) também são valores. A notação $x_!$ na definição de valor de mapa $< \overline{\overline{s_!/v}} >$ define que cada chave de um valor de mapa é única, ou seja, $< \overline{\overline{s_!/v}} >=< s_1/v_1\ s_2/v_2\ s_3/v_3 ...\ s_n/v_n > \wedge \forall i, j \in \{1, 2, 3, ..., n\} : s_i = s_j \implies i = j$.

Um ambiente é uma estrutura de dados que abriga a associação entre atributos (ϑ) e valores (v). São definidas duas operações sobre ambientes: consulta ($\Delta[\![\vartheta]\!]$) e extensão ($\Delta[\overline{\overline{\vartheta/v}}]$). A notação ($\Delta[\![\vartheta]\!]$) denota o valor associado ao atributo ϑ no ambiente Δ. A notação ($\Delta[\vartheta_1/v_1\,\vartheta_2/v_2\,\vartheta_3/v_3 ...\,\vartheta_n/v_n]$ tal que $n \geq 1$) denota a amarração de cada atributo ($\vartheta_1,\ \vartheta_2,\ \vartheta_3, ...,\ \vartheta_n$) ao seu respectivo valor ($v_1,\ v_2,\ v_3, ...,\ v_n$) no ambiente Δ.

```
1 (define-extended-language vAttributeL AttributeLType
2   [v v1 v2 ::= b
3      i
4      s
5      (⇒ (s_!_ v) ...)
6      (: v v)
7      nil]
8      [Δ Δ1 Δ2 ::= ((ϑ_!_ v)...)])
```

Figura 4.8: Especificação em PLT Redex da linguagem *vAttributeL*, que define as sintaxes de valor e ambiente.

A Figura 4.8 apresenta a especificação em Redex da linguagem *vAttributeL*, uma extensão da linguagem *AttributeLType*, especificando valor e ambiente. Os conceitos de valor e ambiente são de suma importância para o desenvolvimento de regras que compõem as semânticas *big-step* e *small-step*, por isso, serão reiteradamente utilizados nas Seções 4.3 e 4.4.

4.3 Semântica *big-step* de PEGwSA

Esta seção descreve uma semântica *big-step* para PEGwSA. As regras que constituem a semântica *big-step* foram divididas em duas partições: uma que trata expressões de atributos (Subseção 4.3.1) e outra que, empregando a primeira, trata expressões de *parsing* (Subseção 4.3.2).

4.3.1 Semântica *big-step* de expressões de atributos

$$\boxed{\Delta \vdash e \Rightarrow v}$$

$$\frac{}{\Delta \vdash b \Rightarrow b}\ \textsc{Booleano} \qquad \frac{}{\Delta \vdash i \Rightarrow i}\ \textsc{Inteiro}$$

$$\frac{}{\Delta \vdash s \Rightarrow s}\ \textsc{Cadeia de Caracteres} \qquad \frac{\Delta[\vartheta] = v}{\Delta \vdash \vartheta \Rightarrow v}\ \textsc{Atributo}$$

$$\frac{}{\Delta \vdash \mathbf{nil} \Rightarrow \mathbf{nil}}\ \textsc{Lista Vazia} \qquad \frac{\Delta \vdash e_1 \Rightarrow v_1 \qquad \Delta \vdash e_2 \Rightarrow v_2}{\Delta \vdash e_1 : e_2 \Rightarrow v_1 : v_2}\ \textsc{Lista}$$

$$\frac{}{\Delta \vdash <> \Rightarrow <>}\ \textsc{Mapa Vazio}$$

$$\frac{\Delta \vdash \mathbf{put} < e_1/e'_1 ... \ e_{n-1}/e'_{n-1} > e_n \ e'_n \Rightarrow < \overline{\overline{s/v}}^{m} >}{\Delta \vdash < e_1/e'_1 ... \ e_{n-1}/e'_{n-1} \ e_n/e'_n > \Rightarrow < \overline{\overline{s/v}}^{m} >}\ \textsc{Mapa}$$

Figura 4.9: Semântica *big-step* de literais, construtores e referências a atributos.

O julgamento da partição que trata expressões de atributos tem a forma $\Delta \vdash e \Rightarrow v$, que deve ser interpretada como: a expressão de atributos e quando avaliada no ambiente Δ produz o valor v. A Figura 4.9 apresenta a semântica *big-step* de literais, construtores e referências a atributos. As expressões de atributos tratadas pelas regras Booleano, Inteiro, Cadeia de Caracteres, Lista Vazia e Mapa Vazio são valores por si só e, por isso, não possuem premissa.

A regra Atributo define que uma referência a um atributo é avaliada para o valor ao qual ele (o atributo) está amarrado no ambiente de avaliação. A regra Lista só é aplicável a listas compostas por um ou mais elementos. Essa regra define que uma lista é avaliada para um valor lista, composto pelos valores para os quais cabeça e cauda da lista original são avaliadas.

Por fim, a Figura 4.9 é encerrada pela apresentação da regra Mapa, que, por sua vez, só é aplicável a mapas compostos por ao menos um elemento. Essa regra tira proveito da semântica de **put** para tratar construtores de mapas. Basicamente, o valor produzido pela avaliação de um mapa qualquer é construído através de uma sucessão de inserções de pares chave-valor (e_n/e'_n). Se uma mesma chave surgir mais de uma vez em

$$\boxed{\Delta \vdash e \Rightarrow v}$$

$$\frac{\Delta \vdash e_1 \Rightarrow i_1 \qquad \Delta \vdash e_2 \Rightarrow i_2 \qquad i = i_1 + i_2}{\Delta \vdash e_1 + e_2 \Rightarrow i}\ \textsc{Adição}$$

$$\frac{\Delta \vdash e_1 \Rightarrow i_1 \qquad \Delta \vdash e_2 \Rightarrow i_2 \qquad i = i_1 \times i_2}{\Delta \vdash e_1 \times e_2 \Rightarrow i}\ \textsc{Multiplicação}$$

$$\frac{\Delta \vdash e_1 \Rightarrow i_1 \qquad \Delta \vdash e_2 \Rightarrow i_2 \qquad i = i_1 - i_2}{\Delta \vdash e_1 - e_2 \Rightarrow i}\ \textsc{Subtração}$$

$$\frac{\Delta \vdash e_1 \Rightarrow i_1 \qquad \Delta \vdash e_2 \Rightarrow i_2 \qquad i = i_1 \div i_2}{\Delta \vdash e_1 \div e_2 \Rightarrow i}\ \textsc{Divisão}$$

$$\frac{\Delta \vdash e_1 \Rightarrow \textbf{true} \qquad \Delta \vdash e_2 \Rightarrow b}{\Delta \vdash e_1 \wedge e_2 \Rightarrow b}\ \textsc{Conjunção}_1 \qquad \frac{\Delta \vdash e_1 \Rightarrow \textbf{false}}{\Delta \vdash e_1 \wedge e_2 \Rightarrow \textbf{false}}\ \neg\textsc{Conjunção}_1$$

$$\frac{\Delta \vdash e_1 \Rightarrow \textbf{true}}{\Delta \vdash e_1 \vee e_2 \Rightarrow \textbf{true}}\ \textsc{Disjunção}_1 \qquad \frac{\Delta \vdash e_1 \Rightarrow \textbf{false} \qquad \Delta \vdash e_2 \Rightarrow b}{\Delta \vdash e_1 \vee e_2 \Rightarrow b}\ \neg\textsc{Disjunção}_1$$

$$\frac{\Delta \vdash e \Rightarrow \textbf{false}}{\Delta \vdash \neg e \Rightarrow \textbf{true}}\ \textsc{Negação Lógica} \qquad \frac{\Delta \vdash e \Rightarrow \textbf{true}}{\Delta \vdash \neg e \Rightarrow \textbf{false}}\ \neg\textsc{Negação Lógica}$$

$$\frac{\Delta \vdash e_1 \Rightarrow v \qquad \Delta \vdash e_2 \Rightarrow v}{\Delta \vdash e_1 == e_2 \Rightarrow \textbf{true}}\ \textsc{Igualdade}$$

$$\frac{\Delta \vdash e_1 \Rightarrow v_1 \qquad \Delta \vdash e_2 \Rightarrow v_2 \qquad v_1 \neq v_2}{\Delta \vdash e_1 == e_2 \Rightarrow \textbf{false}}\ \neg\textsc{Igualdade}$$

$$\frac{\Delta \vdash e_1 \Rightarrow i_1 \qquad \Delta \vdash e_2 \Rightarrow i_2 \qquad i_1 > i_2}{\Delta \vdash e_1 > e_2 \Rightarrow \textbf{true}}\ \textsc{Maior que}$$

$$\frac{\Delta \vdash e_1 \Rightarrow i_1 \qquad \Delta \vdash e_2 \Rightarrow i_2 \qquad i_1 \leq i_2}{\Delta \vdash e_1 > e_2 \Rightarrow \textbf{false}}\ \neg\textsc{Maior que}$$

Figura 4.10: Semântica *big-step* de operações aritméticas, lógicas e relacionais.

uma mesma instância de construtor de mapa, a regra Mapa define que ela (a chave) deve ser amarrada ao valor que estiver mais à direita, ou seja, ao valor "mais recente".

A Figura 4.10 apresenta a semântica *big-step* das operações aritméticas, lógicas e relacionais de PEGwSA. A Figura 4.11 conclui a apresentação da partição da semântica *big-step* de PEGwSA, apresentando as regras que atuam sobre manipuladores de listas e

$$\boxed{\Delta \vdash e \Rightarrow v}$$

$$\frac{\Delta \vdash e \Rightarrow v_1 : v_2}{\Delta \vdash \textbf{head}\ e \Rightarrow v_1}\ \textsc{Head} \qquad \frac{\Delta \vdash e \Rightarrow v_1 : v_2}{\Delta \vdash \mathit{tail}\ e \Rightarrow v_2}\ \textsc{Tail}$$

$$\frac{\Delta \vdash e_1 \Rightarrow < \overline{\overline{s_!/v}} > \qquad \Delta \vdash e_2 \Rightarrow s' \qquad < \overline{\overline{s/v}} > [\![s']\!] = v'}{\Delta \vdash \textbf{get}\ e_1\ e_2 \Rightarrow v'}\ \textsc{Get}$$

$$\frac{\Delta \vdash e_1 \Rightarrow < \overline{\overline{s_!/v}} > \qquad \Delta \vdash e_2 \Rightarrow s' \qquad \Delta \vdash e_3 \Rightarrow v'}{\Delta \vdash \textbf{put}\ e_1\ e_2\ e_3 \Rightarrow < \overline{\overline{s/v}} > [s'/v']}\ \textsc{Put}$$

Figura 4.11: Semântica *big-step* de manipulações de listas e mapas.

mapas.

A regra Head define a semântica da operação de consulta à cabeça de lista (**head**), enquanto a regra Tail define a semântica da operação de consulta à cauda de lista (**tail**). Essas regras implicam implicitamente que manipuladores de listas só podem ser aplicados a listas compostas por um ou mais elementos. Finalmente, são apresentadas as semânticas *big-step* dos manipuladores de mapa responsáveis por consulta (**get**) e inserção (**put**) de elementos em mapas.

4.3.2 Semântica *big-step* de expressões de *parsing*

$$\begin{array}{l} \phi ::= \bar{\bar{t}} \\ w ::= \bar{t} \\ o ::= \bot \mid w \end{array}$$

Figura 4.12: Sintaxe abstrata de entrada e resultado.

A Figura 4.12 define que os símbolos ϕ e w denotam sequências de terminais, esse último requer que a sequência representada esteja limitada por parênteses. O símbolo o denota o resultado da aplicação de expressão de *parsing*, que pode ser falha ($\bot$) ou sucesso (w), em que w, nesse contexto, é a porção da entrada consumida pela aplicação.

O julgamento da partição que trata expressões de *parsing* tem a forma $(G \quad \Delta) \vdash p, w \Rightarrow o \vdash \Delta'$ e deve ser interpretado como: quando avaliada sobre a PEGwSA G e o

ambiente Δ, a aplicação da expressão de *parsing* p à entrada w resulta em o e produz o ambiente Δ'.

$$\boxed{(G \quad \Delta) \vdash p, w \Rightarrow o \vdash \Delta'}$$

$$\frac{}{(G \quad \Delta) \vdash \varepsilon, w \Rightarrow () \vdash \Delta}\ \textsc{Cadeia Vazia} \qquad \frac{}{(G \quad \Delta) \vdash t, (t\ \phi) \Rightarrow (t) \vdash \Delta}\ \textsc{Terminal}$$

$$\frac{t \neq t_1}{(G \quad \Delta) \vdash t, (t_1\ \phi) \Rightarrow \bot \vdash \Delta}\ \neg\textsc{Terminal}_1 \qquad (G \quad \Delta) \vdash t, () \Rightarrow \bot \vdash \Delta\ \neg\textsc{Terminal}_2$$

$$\frac{(G \quad \Delta) \vdash p_1, (\phi_1\ \phi_2\ \phi_3) \Rightarrow (\phi_1) \vdash \Delta_1 \qquad (G \quad \Delta_1) \vdash p_2, (\phi_2\ \phi_3) \Rightarrow (\phi_2) \vdash \Delta_2}{(G \quad \Delta) \vdash p_1 \bullet p_2, (\phi_1\ \phi_2\ \phi_3) \Rightarrow (\phi_1\ \phi_2) \vdash \Delta_2}\ \textsc{Sequência}$$

$$\frac{(G \quad \Delta) \vdash p_1, (\phi_1) \Rightarrow \bot \vdash \Delta_1}{(G \quad \Delta) \vdash p_1 \bullet p_2, (\phi_1) \Rightarrow \bot \vdash \Delta}\ \neg\textsc{Sequência}_1$$

$$\frac{(G \quad \Delta) \vdash p_1, (\phi_1\ \phi_2) \Rightarrow (\phi_1) \vdash \Delta_1 \qquad (G \quad \Delta_1) \vdash p_2, (\phi_2) \Rightarrow \bot \vdash \Delta_2}{(G \quad \Delta) \vdash p_1 \bullet p_2, (\phi_1\ \phi_2) \Rightarrow \bot \vdash \Delta}\ \neg\textsc{Sequência}_2$$

$$\frac{(G \quad \Delta) \vdash p_1, (\phi_1\ \phi_2) \Rightarrow (\phi_1) \vdash \Delta'}{(G \quad \Delta) \vdash p_1/p_2, (\phi_1\ \phi_2) \Rightarrow (\phi_2) \vdash \Delta'}\ \textsc{Escolha Ordenada}_1$$

$$\frac{\begin{array}{c}(G \quad \Delta) \vdash p_1, (\phi_1\ \phi_2) \Rightarrow \bot \vdash \Delta_1 \\ (G \quad \Delta) \vdash p_2, (\phi_1\ \phi_2) \Rightarrow (\phi_1) \vdash \Delta_2\end{array}}{(G \quad \Delta) \vdash p_1/p_2, (\phi_1\ \phi_2) \Rightarrow (\phi_1) \vdash \Delta_2}\ \textsc{Escolha Ordenada}_2$$

$$\frac{(G \quad \Delta) \vdash p_1, w \Rightarrow \bot \vdash \Delta_1 \qquad (G \quad \Delta) \vdash p_2, w \Rightarrow \bot \vdash \Delta_2}{(G \quad \Delta) \vdash p_1/p_2, w \Rightarrow \bot \vdash \Delta}\ \neg\textsc{Escolha Ordenada}$$

Figura 4.13: Semântica *big-step* de cadeia vazia, terminal, sequência e escolha ordenada.

A Figura 4.13 apresenta as regras semânticas que tratam cadeias vazias, terminais, sequências e escolhas ordenadas. Como definido pela regra Cadeia Vazia, cadeias vazias sempre resultam em sucesso sem consumir a entrada nem alterar o ambiente. Um terminal resulta em sucesso se é equivalente ao primeiro símbolo da entrada e falha caso contrário. Terminais, análogas às cadeias vazias, não afetam o ambiente.

Uma sequência resulta em sucesso se a aplicação sucessiva de suas subexpressões (propagando alterações realizadas sobre entrada e ambiente) também resulta em sucesso, conforme definido pela regra Sequência, e falha se a execução de alguma de suas subexpressões falha, como definido pelas regras ¬Sequência$_1$ e ¬Sequência$_2$.

Uma escolha ordenada resulta em sucesso se a aplicação de alguma de suas alternativas resulta em sucesso, como definido pelas regras ESCOLHA ORDENADA$_1$ e ESCOLHA ORDENADA$_2$, e falha se ambas suas alternativas falham (regra ¬ESCOLHA ORDENADA). A segunda alternativa só é executada se a aplicação da primeira falha, por isso essa operação é denominada escolha ordenada.

$$\boxed{(G \quad \Delta) \vdash p, w \Rightarrow o \vdash \Delta'}$$

$$\frac{\begin{array}{c}(G \quad \Delta) \vdash p, (\phi_1\ \phi_2\ \phi_3) \Rightarrow (\phi_1) \vdash \Delta_1 \\ (G \quad \Delta_1) \vdash p*, (\phi_2\ \phi_3) \Rightarrow (\phi_2) \vdash \Delta_2\end{array}}{(G \quad \Delta) \vdash p*, (\phi_1\ \phi_2\ \phi_3) \Rightarrow (\phi_1\ \phi_2) \vdash \Delta_2}\ \text{REPETIÇÃO}_1$$

$$\frac{(G \quad \Delta) \vdash p, w \Rightarrow \bot \vdash \Delta'}{(G \quad \Delta) \vdash p*, w \Rightarrow \varepsilon \vdash \Delta}\ \text{REPETIÇÃO}_2 \qquad \frac{(G \quad \Delta) \vdash p, w \Rightarrow \bot \vdash \Delta'}{(G \quad \Delta) \vdash !p, w \Rightarrow \varepsilon \vdash \Delta'}\ \text{NEGAÇÃO}$$

$$\frac{(G \quad \Delta) \vdash p, (\phi_1\ \phi_2) \Rightarrow (\phi_1) \vdash \Delta'}{(G \quad \Delta) \vdash !p, (\phi_1\ \phi_2) \Rightarrow \bot \vdash \Delta'}\ \neg\text{NEGAÇÃO} \qquad \frac{\Delta \vdash e \Rightarrow v}{(G \quad \Delta) \vdash \vartheta \leftarrow e, w \Rightarrow \varepsilon \vdash \Delta[\vartheta/v]}\ \text{UPDATE}$$

$$\frac{(G \quad \Delta) \vdash p, (\phi_1\ \phi_2) \Rightarrow (\phi_1) \vdash \Delta'}{(G \quad \Delta) \vdash \vartheta = p, (\phi_1\ \phi_2) \Rightarrow \varepsilon \vdash \Delta'[\vartheta/\phi_1]}\ \text{BIND} \qquad \frac{(G \quad \Delta) \vdash p, w \Rightarrow \bot \vdash \Delta'}{(G \quad \Delta) \vdash \vartheta = p, w \Rightarrow \bot \vdash \Delta}\ \neg\text{BIND}$$

$$\frac{\Delta \vdash e \Rightarrow \textbf{true}}{(G \quad \Delta) \vdash ?e, w \Rightarrow \varepsilon \vdash \Delta}\ \text{RESTRIÇÃO} \qquad \frac{\Delta \vdash e \Rightarrow \textbf{false}}{(G \quad \Delta) \vdash ?e, w \Rightarrow \bot \vdash \Delta}\ \neg\text{RESTRIÇÃO}$$

Figura 4.14: Semântica *big-step* de repetição, negação, *update*, *bind* e restrição.

A Figura 4.14 apresenta as semânticas de repetições, negações, *updates*, *binds* e restrições. Executar uma repetição equivale a repetidamente aplicar sua subexpressão, propagando alterações realizadas sobre entrada e ambiente, como definido pela regra REPETIÇÃO$_1$. A regra REPETIÇÃO$_2$ define o condição de parada para a execução da repetição: que ocorre quando sua subexpressão falha.

Uma negação resulta em sucesso sem consumir a entrada se sua subexpressão falha (regra NEGAÇÃO) e falha se sua subexpressão é bem-sucedida (regra ¬NEGAÇÃO). Independentemente do resultado, negações sempre propagam alterações realizadas no ambiente.

A regra UPDATE define que um *update* associa seu atributo ao valor para o qual sua expressão operando é avaliada. Um *bind* amarra seu atributo à cadeia de caracteres

que encerra a porção da entrada consumida por sua subexpressão. Um *bind* resulta em sucesso se a execução de sua subexpressão é bem-sucedida (regra BIND) e falha se a aplicação de sua subexpressão falha (regra ¬BIND). Uma restrição resulta em sucesso se sua expressão operando é avaliada para **true** (regra RESTRIÇÃO) e falha se sua expressão operando é avaliada para **false** (regra ¬RESTRIÇÃO).

$$\boxed{(G \quad \Delta) \vdash p, w \Rightarrow o \vdash \Delta'}$$

$$\frac{\begin{array}{c} G[\![N]\!] = \overline{\vartheta' :: \tau}^n \, \overline{e'}^m \, p \\ \Delta \vdash e_i \Rightarrow v_i, 1 \leq i \leq n \\ (G \quad \overline{\vartheta'/v}^n) \vdash p, (\phi_1 \; \phi_2) \Rightarrow (\phi_1) \vdash \Delta' \\ \Delta' \vdash e'_j \Rightarrow v'_j, 1 \leq j \leq m \end{array}}{(G \quad \Delta) \vdash N \, \overline{e}^n \, \overline{\vartheta}^m, (\phi_1 \; \phi_2) \Rightarrow (\phi_1) \vdash \Delta'[\vartheta_1/v'_1 \; \vartheta_2/v'_2 ... \; \vartheta_m/v'_m]} \text{ CHAMADA DE NÃO-TERMINAL}$$

$$\frac{\begin{array}{c} G[\![N]\!] = \overline{\vartheta' :: \tau}^n \, \overline{e'}^m \, p \\ \Delta \vdash e_i \Rightarrow v_i, 1 \leq i \leq n \\ (G \quad \overline{\vartheta'/v}^n) \vdash p, w \Rightarrow \bot \vdash \Delta' \end{array}}{(G \quad \Delta) \vdash N \, \overline{e}^n \, \overline{\vartheta}^m, w \Rightarrow \bot \vdash \Delta} \text{ ¬CHAMADA DE NÃO-TERMINAL}$$

Figura 4.15: Semântica *big-step* de não-terminal.

Encerrando a apresentação das regras que compõem a semântica *big-step* de PEGwSA, a Figura 4.15 retrata as regras que atuam sobre chamadas de não-terminais.

Antes de explicar o funcionamento dessas regras, é necessário se atentar aos componentes de chamadas de não-terminais e regras de produção. Uma chamada de não-terminal é uma tripla $N \, \overline{e}^n \, \overline{\vartheta}^m$, em que N é o não terminal, $\overline{e}^n$ é a sequência de expressões passadas com parâmetros, e $\overline{\vartheta}^m$ é a sequência de atributos herdados.

Uma regra de produção é uma associação entre um não-terminal N e uma tripla $\overline{\vartheta' :: \tau}^n \, \overline{e'}^m \, p$, na qual o primeiro elemento ($\overline{\vartheta' :: \tau}^n$) é uma sequência de atributos herdades ladeados pelos seus respectivos tipos, $\overline{e'}^m$ é uma sequência de expressões retornadas e p, o corpo da regra de produção.

A semântica de chamada de não-terminal é semelhante a semântica de chamada de função. Primeiramente, é consultado na gramática a regra de produção correspondente à chamada de não-terminal. Em seguida, as expressões passadas como parâmetros são avaliadas para valores. Esses valores são associados aos atributos herdados, formando o

escopo da chamada de não-terminal. O corpo da regra de produção é executado dentro do escopo da chamada de não-terminal, produzindo um novo ambiente. Sobre esse novo ambiente, as expressões retornadas são avaliadas para valores, os quais são associadas aos atributos sintetizados. Essas associações entre atributos sintetizados e valores integram o ambiente no qual a chamada de não-terminal foi efetuada, finalizando o processo de execução da chamada de não-terminal.

A regra Chamada de Não-terminal define que uma chama de não-terminal resulta em sucesso se a aplicação do corpo da regra de produção é bem-sucedida, e falha se essa aplicação falha, conforme definido pela regra ¬Chamada de Não-terminal.

4.4 Semântica *small-step* de PEGwSA

Essa seção descreve uma semântica *small-step* para PEGwSA. As regras de transição que constituem essa semântica *small-step* são apresentadas em duas subseções: a Subseção 4.4.1 reúne as regras que tratam expressões de atributos, já a Subseção 4.4.2, as regras que atuam sobre expressões de *parsing*.

$$
\begin{array}{rcl}
C & ::= & (\blacksquare/p) \mid (p/\blacksquare) \mid (\blacksquare \bullet p) \mid (p \bullet \blacksquare) \mid (\blacksquare *) \mid (!\blacksquare) \mid (?\blacksquare\ e) \mid (\vartheta = \blacksquare\ w) \\
& & \mid (\bigtriangledown \overline{\overline{p}} \bigcirc p \bigtriangleup \overline{\overline{p}} \circ p) \mid (\bigcirc p \bigtriangleup \overline{\overline{p}} \circ p) \mid (\bigtriangleup \overline{\overline{p}} \circ p) \mid (\vartheta \leftarrow e \bigtriangledown) \mid (\vartheta \leftarrow e \bigcirc) \\
& & \mid (\vartheta \leftarrow e \bigtriangleup) \\
E & ::= & E + e \mid v + E \mid E - e \mid v - E \mid E \times e \mid v \times E \mid E \div e \mid v \div E \\
& & \mid E \wedge e \mid E \vee e \mid \neg E \mid E == e \mid v == E \mid E > e \mid v > E \\
& & \mid < \overline{\overline{s/v}}\ E/e\ \overline{\overline{e/e}} > \mid < \overline{\overline{s/v}}\ s/E\ \overline{\overline{e/e}} > \mid \textbf{get}\ E\ e \mid \textbf{get}\ v\ E \mid \textbf{put}\ E\ e\ e \\
& & \mid \textbf{put}\ v\ E\ e \mid \textbf{put}\ v\ s\ E \mid E : e \mid v : E \mid \textbf{head}\ E \mid \textbf{tail}\ E \mid ?E \mid \vartheta \leftarrow E \\
& & \mid \textbf{hole} \\
D & ::= & \downarrow \mid \square \mid \uparrow \\
R & ::= & \bot \mid \top \\
ST & ::= & (G \quad \overline{\Delta}) \vdash \overline{C}\ p\ D\ w\ w'\ R\ \overline{n}
\end{array}
$$

Figura 4.16: Sintaxe abstrata de contextos, estágio de avaliação e resultado.

A Figura 4.16 apresenta a sintaxe de elementos exclusivamente utilizados para descrever as regras de transição da semântica *small-step*, mas que não constituem a sintaxe de PEGwSA em si.

Para compreender o funcionamento da semântica *small-step*, é preciso entender a estrutura de um estado (ST) assim como o papel que cada um de seus componentes

desempenha. Um estado é uma tupla de nove elementos: $(G \quad \overline{\Delta}) \vdash \overline{C}\ p\ D\ w\ w'\ R\ \overline{n}$. O primeiro componente de um estado, G, é uma PEGwSA, que é um conjunto de regras de produção. Uma regra de produção é um mapeamento de um não-terminal para uma tripla $\overline{\vartheta :: \tau}\ \overline{e} \to p$. O primeiro elemento dessa tripla, $\overline{\vartheta :: \tau}$, é uma sequência de atributos aos quais as expressões passadas como parâmetros em uma chamada de não-terminal são associadas, com o intuito de criar um novo ambiente sobre o qual o corpo da regra de produção é executado. O corpo da regra de produção, uma expressão de *parsing* (p), é o último elemento da tripla mencionada. O elemento que entremeia a tripla, $\overline{e}$, é a sequência de expressões retornadas por uma chamada de não-terminal. Portanto, uma PEGwSA (G) armazena informações imprescindíveis à semântica, utilizadas na execução de chamadas de não-terminais.

Logo após a posição que a PEGwSA ocupa no estado, encontra-se a pilha de ambientes ($\overline{\Delta}$). Por via de regra, as expressões de atributos são executadas sobre o ambiente no topo da pilha de ambientes, sendo as expressões passadas como parâmetros em chamadas de não-terminais a única exceção a essa regra. As regras de transição empilham ambientes no estado a fim de garantir a adequada propagação de alterações no ambiente e auxiliar na execução de chamadas de não-terminais (que, por definição, criam um novo ambiente sobre o qual o corpo do não-terminal é executado).

O terceiro elemento que compõe o estado é uma pilha de contextos de expressões de *parsing* ($\overline{C}$). Análogo à relação entre expressões de atributos e pilhas de ambientes, as expressões de *parsing* são executadas sobre o contexto no topo da pilha de contextos. Um contexto de expressão de *parsing* maneja informações que auxiliam a semântica *small-step* a adentrar e retroceder em expressões de *parsing* ao longo de sua execução.

Os próximos quatro componentes de estado, apesar de relativamente mais simples, são tão importantes quanto os já mencionados. O símbolo p, como sabemos, denota uma expressão de *parsing*, mais especificamente a expressão de *parsing* enfoque do estado em questão que compõe. Utilizamos D para denotar o estágio do estado, um estado pode ser classificado em três estágios: estágio de entrada ($\downarrow$), que abriga os estados na eminência de executar a expressão de *parsing* em destaque; estágio de avaliação ($\square$), que reúne os estados sobre os quais as regras de transição que tratam expressões de atributos podem

operar; e estágio de saída ($\uparrow$), que encerra os estados nos quais a expressão de *parsing* em destaque foi executada, e o resultado dessa execução encontra-se em R. O símbolo R, como sugerido, denota o resultado da execução de uma expressão de *parsing*, que pode ser sucesso ($\top$) ou falha ($\bot$). Os subsequentes símbolos w e w' denotam um mesmo tipo de estrutura: uma cadeia de símbolos terminais. O termo w abriga a porção da entrada ainda não consumida, e w', sua contraparte: a porção da entrada que já foi consumida. Essas cadeias de símbolos terminais auxiliam a semântica a desfazer modificações realizadas na entrada sobre a qual a expressão de *parsing* é executada.

Por fim, apresentamos a pilha de números naturais ($\overline{n}$): elemento que encerra a tupla que constitui um estado. De maneira simplificada, é possível afirmar que o topo da pilha geralmente informa a quantidade de símbolos consumidos pela expressão de *parsing* em destaque no estado. As regras de transição que tratam expressões de *parsing* usam dessa pilha adjuntas das cadeias de símbolos terminais, w e w', para restaurar a entrada, ou mais especificamente desfazer as alterações na entrada que uma determinada expressão de *parsing* realizou.

Além da estrutura de estado, dois outros conceitos precisam estar esclarecidos para que seja possível compreender as regras de transição da semântica *small-step*, são eles: 1) por convenção, definimos que o topo da pilha é o seu elemento mais à esquerda; 2) um contexto de expressão de atributos (E), único elemento apresentado na Figura 4.16 que não agrega à estrutura de estado, é fundamental às regras de transição que manejam estados no estágio de avaliação, com objetivo de avaliar uma expressão de atributos para um valor.

Diferente da semântica *big-step*, que possui julgamentos distintos para expressões de atributos e expressões de *parsing*, a semântica *small-step* possui um único julgamento, que tem a forma $ST \Rightarrow ST'$, ou seja, as regras que compõem a semântica *small-step* retratam a transição de um estado para outro, à vista disso escolhemos endereçá-las como regras de transição.

4.4.1 Semântica *small-step* de expressões de atributos

Um contexto (ŞERBăNUţă; ROşU; MESEGUER, 2009) é um programa com um "*hole*", sendo esse *hole* um espaço reservado sobre o qual o próximo passo computacional ocorre.

Um contexto de avaliação (PIERCE, 2002) é um termo com um buraco dentro. Contextos de avaliação capturam a noção de "próximo subtermo a ser reduzido".

Seguindo o mesmo conceito de contexto retratado por Pierce (2002), Şerbănuţă, Roşu e Meseguer (2009), um contexto de expressão de atributos é uma expressão de atributos com um buraco, sendo esse buraco uma expressão de atributos em si. O próximo passo da avaliação da expressão de atributos se dá através da substituição desse buraco por uma expressão de atributos equivalente. A sintaxe de contexto de expressão de atributos, retratada na Figura 4.16, define as condições/posições em que um buraco pode ocorrer e uma direção de avaliação de expressões de atributos: da esquerda para a direita. Todas as regras de transição que tratam expressões de atributos são aplicadas dentro de buracos de contextos de expressões de atributos.

$$\boxed{(G \quad \overline{\Delta}) \vdash \overline{C}\ p\ D\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta'}) \vdash \overline{C'}\ p'\ D'\ w''\ w'''\ R'\ \overline{n'}}$$

$$\frac{i' = i_1 + i_2}{(G \quad \overline{\Delta}) \vdash \overline{C}\ E[i_1 + i_2] \square\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta}) \vdash \overline{C}\ E[i'] \square\ w\ w'\ R\ \overline{n}}\ \textsc{Adição}$$

$$\frac{i' = i_1 - i_2}{(G \quad \overline{\Delta}) \vdash \overline{C}\ E[i_1 - i_2] \square\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta}) \vdash \overline{C}\ E[i'] \square\ w\ w'\ R\ \overline{n}}\ \textsc{Subtração}$$

$$\frac{i' = i_1 \times i_2}{(G \quad \overline{\Delta}) \vdash \overline{C}\ E[i_1 \times i_2] \square\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta}) \vdash \overline{C}\ E[i'] \square\ w\ w'\ R\ \overline{n}}\ \textsc{Multiplicação}$$

$$\frac{i' = i_1 \div i_2}{(G \quad \overline{\Delta}) \vdash \overline{C}\ E[i_1 \div i_2] \square\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta}) \vdash \overline{C}\ E[i'] \square\ w\ w'\ R\ \overline{n}}\ \textsc{Divisão}$$

Figura 4.17: Semântica *small-step* de operações aritméticas.

A Figura 4.17 apresenta as regras de transição que tratam operações aritméticas, tais regras só podem ser aplicadas quando ambos operandos são literais inteiros. Em seguida, na Figura 4.18, são apresentadas as regras de transição que atuam sobre operações lógicas, que só podem ser aplicadas quando o primeiro operando é um literal booleano.

$$\boxed{(G \quad \overline{\Delta}) \vdash \overline{C}\ p\ D\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta'}) \vdash \overline{C'}\ p'\ D'\ w''\ w'''\ R'\ \overline{n'}}$$

$$\frac{}{(G \quad \overline{\Delta}) \vdash \overline{C}\ E[\mathbf{true} \wedge e]\ \square\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta}) \vdash \overline{C}\ E[e]\ \square\ w\ w'\ R\ \overline{n}}\ \textsc{Conjunção}_1$$

$$\frac{}{(G \quad \overline{\Delta}) \vdash \overline{C}\ E[\mathbf{false} \wedge e]\ \square\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta}) \vdash \overline{C}\ E[\mathbf{false}]\ \square\ w\ w'\ R\ \overline{n}}\ \neg\textsc{Conjunção}_1$$

$$\frac{}{(G \quad \overline{\Delta}) \vdash \overline{C}\ E[\mathbf{true} \vee e]\ \square\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta}) \vdash \overline{C}\ E[\mathbf{true}]\ \square\ w\ w'\ R\ \overline{n}}\ \textsc{Disjunção}_1$$

$$\frac{}{(G \quad \overline{\Delta}) \vdash \overline{C}\ E[\mathbf{false} \vee e]\ \square\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta}) \vdash \overline{C}\ E[e]\ \square\ w\ w'\ R\ \overline{n}}\ \neg\textsc{Disjunção}_1$$

$$\frac{}{\begin{array}{c}(G \quad \overline{\Delta}) \vdash \overline{C}\ E[\neg\ \mathbf{true}]\ \square\ w\ w'\ R\ \overline{n} \Rightarrow \\ (G \quad \overline{\Delta}) \vdash \overline{C}\ E[\mathbf{false}]\ \square\ w\ w'\ R\ \overline{n}\end{array}}\ \textsc{Negação Lógica}$$

$$\frac{}{\begin{array}{c}(G \quad \overline{\Delta}) \vdash \overline{C}\ E[\neg\ \mathbf{false}]\ \square\ w\ w'\ R\ \overline{n}^c \Rightarrow \\ (G \quad \overline{\Delta}) \vdash \overline{C}\ E[\mathbf{true}]\ \square\ w\ w'\ R\ \overline{n}\end{array}}\ \neg\textsc{Negação Lógica}$$

Figura 4.18: Semântica *small-step* de operações lógicas.

$$\boxed{(G \quad \overline{\Delta}) \vdash \overline{C}\ p\ D\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta'}) \vdash \overline{C'}\ p'\ D'\ w''\ w'''\ R'\ \overline{n'}}$$

$$\frac{}{(G \quad \overline{\Delta}) \vdash \overline{C}\ E[v == v]\ \square\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta}) \vdash \overline{C}\ E[\mathbf{true}]\ \square\ w\ w'\ R\ \overline{n}}\ \textsc{Iguadade}$$

$$\frac{v_1 \neq v_2}{(G \quad \overline{\Delta}) \vdash \overline{C}\ E[v_1 == v_2]\ \square\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta}) \vdash \overline{C}\ E[\mathbf{false}]\ \square\ w\ w'\ R\ \overline{n}}\ \neg\textsc{Iguadade}$$

$$\frac{i_1 > i_2}{(G \quad \overline{\Delta}) \vdash \overline{C}\ E[i_1 > i_2]\ \square\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta}) \vdash \overline{C}\ E[\mathbf{true}]\ \square\ w\ w'\ R\ \overline{n}}\ \textsc{Maior que}$$

$$\frac{i_1 \leq i_2}{(G \quad \overline{\Delta}) \vdash \overline{C}\ E[i_1 > i_2]\ \square\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta}) \vdash \overline{C}\ E[\mathbf{false}]\ \square\ w\ w'\ R\ \overline{n}}\ \neg\textsc{Maior que}$$

Figura 4.19: Semântica *small-step* de operações relacionais.

Como as regras de transição que tratam operações lógicas binárias (conjunção e disjunção) podem ser aplicados mesmo quando o segundo operando é uma expressão de atributos qualquer, um contexto de expressão de atributos não pode ser $v \wedge E$ nem $v \vee E$, tendo em vista que esses termos adjuntos das atuais regras de transição introduzem ambiguidade na semântica *small-step* de PEGwSA. A Figura 4.19 apresenta as regras de transição que

atuam sobre operações relacionais. Análogas às regras que tratam operações aritméticas, essas regras de transição somente exigem que ambos os operandos sejam literais inteiros.

$$\boxed{(G \quad \overline{\Delta}) \vdash \overline{C}\ p\ D\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta'}) \vdash \overline{C'}\ p'\ D'\ w''\ w'''\ R'\ \overline{n'}}$$

$$\frac{}{(G \quad \overline{\Delta}) \vdash \overline{C}\ E[\mathbf{head}\ v_1 : v_2]\ \square\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta}) \vdash \overline{C}\ E[v_1]\ \square\ w\ w'\ R\ \overline{n}}\ \textsc{Head}$$

$$\frac{}{(G \quad \overline{\Delta}) \vdash \overline{C}\ E[\mathbf{tail}\ v_1 : v_2]\ \square\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta}) \vdash \overline{C}\ E[v_2]\ \square\ w\ w'\ R\ \overline{n}}\ \textsc{Tail}$$

Figura 4.20: Semântica *small-step* de manipuladores de lista.

$$\boxed{(G \quad \overline{\Delta}) \vdash \overline{C}\ p\ D\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta'}) \vdash \overline{C'}\ p'\ D'\ w''\ w'''\ R'\ \overline{n'}}$$

$$\frac{\begin{array}{c}\exists i, j \in \mathbb{N}^* : 1 \leq i, j \leq a \wedge i \neq j \wedge s_i = s_j \\ p_1 =< s_1/v_1 > \qquad p_k = \mathbf{put}\ p_{k-1}\ s_k\ v_k, 1 < k \leq a\end{array}}{(G \quad \overline{\Delta}) \vdash \overline{C}\ E[< \overline{\overline{s/v}}^{a} >]\ \square\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta}) \vdash \overline{C}\ E[p_a]\ \square\ w\ w'\ R\ \overline{n}}\ \textsc{Mapa}$$

$$\frac{< \overline{\overline{s_!/v}} > [\![s']\!] = v'}{(G \quad \overline{\Delta}) \vdash \overline{C}\ E[\mathbf{get}\ < \overline{\overline{s_!/v}} >\ s']\ \square\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta}) \vdash \overline{C}\ E[v']\ \square\ w\ w'\ R\ \overline{n}}\ \textsc{Get}$$

$$\frac{}{\begin{array}{c}(G \quad \overline{\Delta}) \vdash \overline{C}\ E[\mathbf{put}\ < \overline{\overline{s_!/v}} >\ s'\ v']\ \square\ w\ w'\ R\ \overline{n} \Rightarrow \\ (G \quad \overline{\Delta}) \vdash \overline{C}\ E[< \overline{s_!/v} > [s'/v']]\ \square\ w\ w'\ R\ \overline{n}\end{array}}\ \textsc{Put}$$

Figura 4.21: Semântica *small-step* de manipuladores de mapeamento.

A Figura 4.20 apresenta as regras de transição que tratam manipuladores de listas. Uma abordagem elegante para tratar manipuladores de listas é operar sobre listas descritas na notação cabeça-cauda sem avaliar suas subexpressões para valores, tendo em vista que essa abordagem descarta transições desnecessárias (e.g., é desnecessário avaliar a cauda de uma lista para um valor em uma consulta à cabeça). Entretanto, as vigentes regras de transição que tratam manipuladores de listas exigem que ambas cabeça e cauda sejam avaliadas para valores, tendo em vista que a outra opção (a abordagem citada logo acima) introduz ambiguidade na semântica *small-step*, uma vez que um contexto de expressão de atributos pode ser **head** E e **tail** E, ou seja, é permitido às regras de transição operar sobre os operandos de manipuladores de listas. É possível alterar as

regras de transição a fim de contornar a introdução de ambiguidade, mas além de deixar mais complexa a semântica *small-step*, essas alterações a afastam do comportamento da semântica *big-step*, característica indesejada na semântica *small-step* almejada.

A Figura 4.21 apresenta as regras de transição que tratam construtores e manipuladores de mapas. A regra MAPA atua sobre construtores de mapas cujas chaves e valores foram previamente avaliadas, respectivamente, para literais cadeias de caracteres e valores e contenham ao menos uma chave repetida. Essa regra tira proveito da operação de inserção de elementos em mapas (**put**) a fim de remover essa ambiguidade. Assumimos que a alteração mais recente de um mapa está mais à direita, ou seja, se uma mesma chave é repetida várias vezes em um mesmo literal mapa, ela (a chave) será amarrada ao valor associado a sua instância mais à direita. As semânticas *small-step* de manipuladores de mapas (consulta e inserção de elementos) são apresentadas em sequência. Ambas só podem ser aplicadas quando ambos os operandos são valores.

$$\boxed{(G \quad \overline{\Delta}) \vdash \overline{C}\ p\ D\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta'}) \vdash \overline{C'}\ p'\ D'\ w''\ w'''\ R'\ \overline{n'}}$$

$$\frac{C \neq (p \bigvee) \qquad \Delta[\![\vartheta]\!] = v}{\begin{array}{c}(G \quad (\Delta\ \overline{\overline{\Delta'}})) \vdash (C\ \overline{\overline{C'}})\ E[\vartheta] \square\ w\ w'\ R\ \overline{n} \Rightarrow \\ (G \quad (\Delta\ \overline{\overline{\Delta'}})) \vdash (C\ \overline{\overline{C'}})\ E[v] \square\ w\ w'\ R\ \overline{n}\end{array}}\ \textsc{Atributo}$$

$$\frac{C = (p \bigvee) \qquad \Delta_2[\![\vartheta]\!] = v}{\begin{array}{c}(G \quad (\Delta_1\ \Delta_2\ \overline{\overline{\Delta'}})) \vdash (C\ \overline{\overline{C'}})\ E[\vartheta] \square\ w\ w'\ R\ \overline{n} \Rightarrow \\ (G \quad (\Delta_1\ \Delta_2\ \overline{\overline{\Delta'}})) \vdash (C\ \overline{\overline{C'}})\ E[v] \square\ w\ w'\ R\ \overline{n}\end{array}}\ \textsc{Parâmetro}$$

Figura 4.22: Semântica *small-step* de referências a atributos.

A Figura 4.22 apresenta as regras de transição que tratam referências a atributos. A regra ATRIBUTO atua sobre referências a atributos que não estão contidas em expressões passadas como parâmetros de chamadas de não-terminais, e, por isso, o valor associado ao atributo é consultado no topo da pilha de ambientes. Em geral, uma expressão de atributos é avaliada sobre o ambiente que encabeça a pilha de ambientes, com expressões passadas como parâmetros de chamadas de não-terminais sendo a única exceção.

Já a regra PARÂMETRO, como o próprio nome sugere, atua sobre referências

a atributos contidos em expressões passadas como parâmetros em chamadas de não-terminais. Por esse motivo, o valor associado ao atributo é consultado no ambiente subsequente ao topo da pilha de ambientes. Isso ocorre porque, nesse contexto, o topo da pilha de ambientes é o ambiente sobre o qual o corpo da chamada de não-terminal será executado, enquanto o ambiente subsequente ao topo da pilha é o ambiente no qual o não-terminal foi chamado. O contexto de expressão de *parsing* no topo da pilha de contextos é consultado para constatar se a referência a atributo está ou não contida em uma expressão de atributos passada como parâmetro em uma chamada de não-terminal. Na Seção 4.4.2, são apresentados as decisões de projeto que motivam este comportamento singular.

4.4.2 Semântica *small-step* de expressões de *parsing*

Essa seção disserta sobre as regras de transição que tratam expressões de *parsing*.

$$\boxed{(G \quad \overline{\Delta}) \vdash \overline{C}\ p\ D\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta'}) \vdash \overline{C'}\ p'\ D'\ w''\ w'''\ R'\ \overline{n'}}$$

$$\frac{n'' = n + 1}{(G \quad \overline{\Delta}) \vdash \overline{C}\ t \downarrow (t\ \overline{\overline{t'}})\ (\overline{\overline{t''}})\ R\ (n\ \overline{\overline{n'}}) \Rightarrow (G \quad \overline{\Delta}) \vdash \overline{C}\ t \uparrow (\overline{\overline{t'}})\ (\overline{\overline{t''}}\ t)\ \top\ (n''\ \overline{\overline{n'}})}\ \downarrow\textsc{Terminal}$$

$$\frac{t \neq t'}{(G \quad \overline{\Delta}) \vdash \overline{C}\ t \downarrow (t'\ \overline{\overline{t''}})\ w\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta}) \vdash \overline{C}\ t \uparrow (t'\ \overline{\overline{t''}})\ w \perp \overline{n}}\ \downarrow \neg\textsc{Terminal}_1$$

$$\frac{}{(G \quad \overline{\Delta}) \vdash \overline{C}\ t \downarrow ()\ w\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta}) \vdash \overline{C}\ t \uparrow ()\ w \perp \overline{n}}\ \downarrow \neg\textsc{Terminal}_2$$

Figura 4.23: Semântica *small-step* de terminal.

A Figura 4.23 apresenta as regras de transição que tratam terminais. Essas regras só podem ser aplicadas a um estado se ele está no estágio de entrada ($\downarrow$) e sua expressão é um terminal, ou seja, se ele está na eminência de executar um terminal. Independentemente de qual das três regras contidas na Figura 4.23 seja aplicada, um novo estado é produzido. Esse novo estado deve estar no estágio de saída ($\downarrow$) e armazenar em R o resultado da aplicação do terminal em pauta. A regra $\downarrow$Terminal define a situação em que a aplicação de um terminal resulta em sucesso: o que ocorre quando a porção da

entrada restante é encabeçada pelo mesmo símbolo que o terminal consome. Quando um terminal resulta em sucesso, é preciso realizar duas alterações específicas no estado: 1) passar o símbolo consumido pelo terminal do início da porção de entrada restante para o fim da porção de entrada consumida; 2) incrementar o topo da pilha de naturais em uma unidade, deixando expresso aos estados subsequentes que mais um símbolo da entrada foi consumido. As regras $\downarrow \neg\textsc{Terminal}_1$ e $\downarrow \neg\textsc{Terminal}_2$ definem, respectivamente, as condições nas quais a aplicação de um terminal falha: quando ocorre disparidade entre o terminal e o símbolo que encabeça a porção da entrada restante e quando a porção da entrada restante é vazia.

$$\boxed{(G \quad \overline{\Delta}) \vdash \overline{C}\ p\ D\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta'}) \vdash \overline{C'}\ p'\ D'\ w''\ w'''\ R'\ \overline{n'}}$$

$$\frac{}{(G \quad \overline{\Delta}) \vdash \overline{C}\ \varepsilon \downarrow w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta}) \vdash \overline{C}\ \varepsilon \uparrow w\ w' \top \overline{n}}\ \downarrow\textsc{Cadeia Vazia}$$

Figura 4.24: Semântica *small-step* de cadeia vazia.

A Figura 4.24 apresenta a única regra de transição que trata cadeias vazias, denominada Cadeia Vazia. Essa regra somente pode ser aplicada a um estado cuja expressão de *parsing* é uma cadeia vazia (ε) e que está no estágio de entrada ($\downarrow$), ou seja, um estado precedente à execução de uma cadeia vazia. O estado produzido pela aplicação da regra Cadeia Vazia diverge do estado sobre o qual a regra foi aplicada em somente dois de seus componentes: o estágio e o resultado, pois o estado produzido deve estar no estágio de saída e sinalizar que a execução da cadeia vazia resultou em sucesso.

A Figura 4.25 apresenta as regras de transição que tratam escolhas ordenadas. Antes de descrever a abordagem que empregamos para ultrapassar os obstáculos encontrados ao desenvolver essas regras, é necessário esclarecer quais são os obstáculos.

O primeiro obstáculo encontra-se no processo de retroceder a entrada a um estado anterior à aplicação de uma expressão de *parsing*. Primeiramente, é preciso se atentar ao fato que a entrada, contida em um estado, é obtida através da concatenação de dois componentes do estado, símbolos consumidos (w') e símbolos restantes (w), que são apresentados na ordem reversa dessa concatenação: $(G \quad \overline{\Delta}) \vdash \overline{C}\ p\ D\ w\ w'\ R\ \overline{n}$.

Somente é necessário retroceder a entrada a um estado anterior à aplicação de uma

$$\boxed{(G \quad \overline{\Delta}) \vdash \overline{C}\ p\ D\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta'}) \vdash \overline{C'}\ p'\ D'\ w''\ w'''\ R'\ \overline{n'}}$$

$$\frac{}{\begin{array}{c}(G \quad (\Delta\ \overline{\overline{\Delta'}})) \vdash (\overline{\overline{C}})\ p_1/p_2 \downarrow w\ w'\ R\ (\overline{\overline{n}}) \Rightarrow \\ (G \quad (\Delta\ \Delta\ \overline{\overline{\Delta'}})) \vdash ((\blacksquare/p_2)\ \overline{\overline{C}})\ p_1 \downarrow w\ w'\ R\ (0\ \overline{\overline{n}})\end{array}}\ \downarrow\textsc{Escolha Ordenada}$$

$$\frac{n'' = n_1 + n_2}{\begin{array}{c}(G \quad (\Delta_1\ \Delta_2\ \overline{\overline{\Delta'}})) \vdash ((\blacksquare/p_2)\ \overline{\overline{C}})\ p_1 \uparrow w\ w' \top (n_1\ n_2\ \overline{\overline{n'}}) \Rightarrow \\ (G \quad (\Delta_1\ \overline{\overline{\Delta'}})) \vdash (\overline{\overline{C}})\ p_1/p_2 \uparrow w\ w' \top (n''\ \overline{\overline{n'}})\end{array}}\ \uparrow\textsc{Escolha Ordenada}_1$$

$$\frac{n \neq 0 \qquad n'' = n - 1}{\begin{array}{c}(G \quad \overline{\Delta}) \vdash ((\blacksquare/p_2)\ \overline{\overline{C}})\ p_1 \uparrow (\overline{\overline{t}})\ (\overline{\overline{t'}}\ t'') \bot (n\ \overline{\overline{n'}}) \Rightarrow \\ (G \quad \overline{\Delta}) \vdash ((\blacksquare/p_2)\ \overline{\overline{C}})\ p_1 \uparrow (t''\ \overline{\overline{t}})\ (\overline{\overline{t'}}) \bot (n''\ \overline{\overline{n'}})\end{array}}\ \textsc{Retroceder Escolha Ordenada}$$

$$\frac{}{\begin{array}{c}(G \quad (\Delta_1\ \Delta_2\ \overline{\overline{\Delta'}})) \vdash ((\blacksquare/p_2)\ \overline{\overline{C}})\ p_1 \uparrow w\ w' \bot (0\ \overline{\overline{n}}) \Rightarrow \\ (G \quad (\Delta_2\ \Delta_2\ \overline{\overline{\Delta'}})) \vdash ((p_1/\blacksquare)\ \overline{\overline{C}})\ p_2 \downarrow w\ w' \bot (0\ \overline{\overline{n}})\end{array}}\ \uparrow\neg\textsc{Escolha Ordenada}_1$$

$$\frac{n'' = n_1 + n_2}{\begin{array}{c}(G \quad (\Delta_1\ \Delta_2\ \overline{\overline{\Delta'}})) \vdash ((p_1/\blacksquare)\ \overline{\overline{C}})\ p_2 \uparrow w\ w' \top (n_1\ n_2\ \overline{\overline{n'}}) \Rightarrow \\ (G \quad (\Delta_1\ \overline{\overline{\Delta'}})) \vdash (\overline{\overline{C}})\ p_1/p_2 \uparrow w\ w' \top (n''\ \overline{\overline{n'}})\end{array}}\ \uparrow\textsc{Escolha Ordenada}_2$$

$$\frac{n'' = n_1 + n_2}{\begin{array}{c}(G \quad (\Delta_1\ \Delta_2\ \overline{\overline{\Delta'}})) \vdash ((p_1/\blacksquare)\ \overline{\overline{C}})\ p_2 \uparrow w\ w' \bot (n_1\ n_2\ \overline{\overline{n'}}) \Rightarrow \\ (G \quad (\Delta_2\ \overline{\overline{\Delta'}})) \vdash (\overline{\overline{C}})\ p_1/p_2 \uparrow w\ w' \bot (n''\ \overline{\overline{n'}})\end{array}}\ \uparrow\neg\textsc{Escolha Ordenada}_2$$

Figura 4.25: Semântica *small-step* de escolha ordenada.

expressão de *parsing* em três ocasiões: 1) quando a execução da primeira alternativa de uma escolha ordenada falha, tendo em vista que a segunda alternativa deve ser executada sobre o mesmo estado de entrada sobre o qual a primeira alternativa foi executada; 2) quando a execução da subexpressão de uma repetição falha, uma vez que as alterações realizadas sobre ambiente e entrada são descartadas nessa situação; 3) quando a execução da subexpressão de uma negação falha, pois negações não consomem nenhum símbolo da entrada quando resultam em sucesso.

A estratégia utilizada para retroceder a entrada a um estado anterior envolve a pilha de números naturais ($\overline{n}$), que, nessas ocasiões, abriga no topo a quantidade de símbolos consumidos pela expressão de *parsing* cujos efeitos pretende-se desfazer. Sendo assim, simplesmente é realizada uma iteração em que, enquanto o topo da pilha de números

naturais for diferente de zero, move o último símbolo da sequência de símbolos consumidos para o início da sequência de símbolos restantes, decrescendo o topo da pilha de números naturais em uma unidade. Quando a iteração termina, a entrada está no estado precedente à execução da expressão de *parsing* em enfoque.

Outro obstáculo encontra-se em desenvolver um mecanismo que permite a propagação e descarte de alterações realizadas no ambiente por uma expressão de *parsing*. A abordagem adotada para contornar esse obstáculo faz uso da pilha de ambientes. Em suma, na eminência de executar uma expressão de *parsing* cujas alterações no ambiente podem ser descartadas, o topo da pilha é duplicado. A expressão de *parsing* em questão é executada sobre essa duplicata. Se a aplicação da expressão de *parsing* resultar em sucesso, a duplicata substitui o ambiente original (que é descartado), propagando as alterações no ambiente. Caso a execução da expressão de *parsing* falhe, a duplicada é descartada e nenhuma alteração no ambiente é propagada.

Dois contextos de expressões de *parsing* são usados para tratar escolhas ordenadas: $(\blacksquare/p_2)$ e $(p_1/\blacksquare)$. O símbolo $\blacksquare$, nesse caso, sinaliza qual alternativa da escolha ordenada está em execução.

A regra ↓Escolha Ordenada prepara o terreno para aplicar a primeira alternativa da escolha ordenada, e a regra ↑ ¬Escolha Ordenada$_1$, sua segunda alternativa. As regras ↑Escolha Ordenada$_1$ e ↑Escolha Ordenada$_2$ definem que uma ecolha ordenada resulta em sucesso, respectivamente, quando sua primeira ou segunda alternativa resultam em sucesso, propagando alterações realizadas no ambiente. A regra ↑ ¬Escolha Ordenada$_2$ define que uma escolha ordenada falha quando ambas suas alternativas falham. Por fim, a regra Retroceder Escolha Ordenada retrocede a entrada ao estado anterior à aplicação da primeira alternativa da escolha ordenada.

A Figura 4.26 apresenta as regras de transição que tratam sequências. Os contextos de expressões de *parsing* $(\blacksquare \bullet p_2)$ e $(p_1 \bullet \blacksquare)$ são usados para sinalizar qual subexpressão da sequência está em execução. A regra ↓Sequência prepara o terreno para a aplicação da primeira subexpressão de uma sequência, e a regra ↑Sequência$_1$, para a sua segunda subexpressão. A regra ↑Sequência$_2$ define que a execução de uma sequência resulta em sucesso se aplicação sucessiva de suas subexpressões resulta em sucesso, propagando as

$$\boxed{(G \quad \overline{\Delta}) \vdash \overline{C}\ p\ D\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta'}) \vdash \overline{C'}\ p'\ D'\ w''\ w'''\ R'\ \overline{n'}}$$

$$\frac{}{\begin{array}{l}(G \quad (\Delta\ \overline{\overline{\Delta'}})) \vdash (\overline{\overline{C}})\ p_1 \bullet p_2 \downarrow w\ w'\ R\ \overline{n} \Rightarrow \\ (G \quad (\Delta\ \Delta\ \overline{\overline{\Delta'}})) \vdash ((\blacksquare \bullet p_2)\ \overline{\overline{C}})\ p_1 \downarrow w\ w'\ R\ \overline{n}\end{array}}\ \downarrow\textsc{Sequência}$$

$$\frac{}{\begin{array}{l}(G \quad \overline{\Delta}) \vdash ((\blacksquare \bullet p_2)\ \overline{\overline{C}})\ p_1 \uparrow w\ w'\ \top\ \overline{n} \Rightarrow \\ (G \quad \overline{\Delta}) \vdash ((p_1 \bullet \blacksquare)\ \overline{\overline{C}})\ p_2 \downarrow w\ w'\ \top\ \overline{n}\end{array}}\ \uparrow\textsc{Sequência}_1$$

$$\frac{}{\begin{array}{l}(G \quad (\Delta\ \overline{\overline{\Delta'}})) \vdash ((\blacksquare \bullet p_2)\ \overline{\overline{C}})\ p_1 \uparrow w\ w'\ \bot\ \overline{n} \Rightarrow \\ (G \quad (\overline{\overline{\Delta'}})) \vdash (\overline{\overline{C}})\ p_1 \bullet p_2 \uparrow w\ w'\ \bot\ \overline{n}\end{array}}\ \uparrow \neg\textsc{Sequência}_1$$

$$\frac{}{\begin{array}{l}(G \quad (\Delta_1\ \Delta_2\ \overline{\overline{\Delta'}})) \vdash ((p_1 \bullet \blacksquare)\ \overline{\overline{C}})\ p_2 \uparrow w\ w'\ \top\ \overline{n} \Rightarrow \\ (G \quad (\Delta_1\ \overline{\overline{\Delta'}})) \vdash (\overline{\overline{C}})\ p_1 \bullet p_2 \uparrow w\ w'\ \top\ \overline{n}\end{array}}\ \uparrow\textsc{Sequência}_2$$

$$\frac{}{\begin{array}{l}(G \quad (\Delta\ \overline{\overline{\Delta'}})) \vdash ((p_1 \bullet \blacksquare)\ \overline{\overline{C}})\ p_2 \uparrow w\ w'\ \bot\ \overline{n} \Rightarrow \\ (G \quad (\overline{\overline{\Delta'}})) \vdash (\overline{\overline{C}})\ p_1 \bullet p_2 \uparrow w\ w'\ \bot\ \overline{n}\end{array}}\ \uparrow \neg\textsc{Sequência}_2$$

Figura 4.26: Semântica *small-step* de sequência.

alterações realizadas no ambiente. As regras $\uparrow \neg$Sequência$_1$ e $\uparrow \neg$Sequência$_2$ definem que uma sequência falha se a aplicação de alguma de suas subexpressões falha.

A Figura 4.27 retrata as regras que atuam sobre repetições. O contexto de expressão de *parsing* ($\blacksquare*$) indica que uma repetição está em execução. A regra $\downarrow$Repetição cria as condições para a aplicação da repetição. A regra $\uparrow$Repetição define que uma repetição é reaplicada quando a execução de sua subexpressão resulta em sucesso. A regra Retroceder Repetição retrocede a entrada ao estado anterior à aplicação da subexpressão de uma repetição. A regra $\uparrow \neg$Repetição define o ponto de parada da aplicação da repetição, que ocorre quando a aplicação de sua subexpressão falha.

A Figura 4.28 apresenta as regras de transição que tratam negações. O contexto de expressão de *parsing* ($!\blacksquare$) sinaliza que uma negação está em execução. A regra $\downarrow$Negação prepara o estado para aplicação de uma negação. Como uma negação, independentemente de resultar em sucesso ou falhar, sempre propaga as alterações realizadas no ambiente, o topo da pilha de ambientes não precisa ser duplicado. A regra Retroceder Negação restaura a entrada ao estado anterior a aplicação da subexpressão da

$$\boxed{(G \quad \overline{\Delta}) \vdash \overline{C}\ p\ D\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta'}) \vdash \overline{C'}\ p'\ D'\ w''\ w'''\ R'\ \overline{n'}}$$

$$\frac{}{\begin{array}{c}(G \quad (\Delta\ \overline{\overline{\Delta'}})) \vdash (\overline{\overline{C}})\ p* \downarrow w\ w'\ R\ (\overline{\overline{n}}) \Rightarrow \\ (G \quad (\Delta\ \Delta\ \overline{\overline{\Delta'}})) \vdash ((\blacksquare*)\ \overline{\overline{C}})\ p \downarrow w\ w'\ R\ (0\ \overline{\overline{n}})\end{array}}\ \downarrow\textsc{Repetição}$$

$$\frac{n'' = n_1 + n_2}{\begin{array}{c}(G \quad (\Delta_1\ \Delta_2\ \overline{\overline{\Delta'}})) \vdash ((\blacksquare*)\ \overline{\overline{C}})\ p \uparrow w\ w' \top (n_1\ n_2\ \overline{\overline{n'}}) \Rightarrow \\ (G \quad (\Delta_1\ \Delta_1\ \overline{\overline{\Delta'}})) \vdash ((\blacksquare*)\ \overline{\overline{C}})\ p \downarrow w\ w' \top (0\ n''\ \overline{\overline{n'}})\end{array}}\ \uparrow\textsc{Repetição}$$

$$\frac{n \neq 0 \qquad n'' = n - 1}{\begin{array}{c}(G \quad \overline{\Delta}) \vdash ((\blacksquare*)\ \overline{\overline{C}})\ p \uparrow (\overline{\bar{t}})\ (\overline{\overline{t'}}\ t'') \bot (n\ \overline{\overline{n'}}) \Rightarrow \\ (G \quad \overline{\Delta}) \vdash ((\blacksquare*)\ \overline{\overline{C}})\ p \uparrow (t''\ \overline{\bar{t}})\ (\overline{\overline{t'}}) \bot (n''\ \overline{\overline{n'}})\end{array}}\ \textsc{Retroceder Repetição}$$

$$\frac{}{\begin{array}{c}(G \quad (\Delta\ \overline{\overline{\Delta'}})) \vdash ((\blacksquare*)\ \overline{\overline{C}})\ p \uparrow w\ w' \bot (0\ \overline{\overline{n}}) \Rightarrow \\ (G \quad (\overline{\overline{\Delta'}})) \vdash (\overline{\overline{C}})\ p* \uparrow w\ w' \top (\overline{\overline{n}})\end{array}}\ \uparrow\neg\textsc{Repetição}$$

Figura 4.27: Semântica *small-step* de repetição.

$$\boxed{(G \quad \overline{\Delta}) \vdash \overline{C}\ p\ D\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta'}) \vdash \overline{C'}\ p'\ D'\ w''\ w'''\ R'\ \overline{n'}}$$

$$\frac{}{(G \quad \overline{\Delta}) \vdash (\overline{\overline{C}})\ !p \downarrow w\ w'\ R\ (\overline{\overline{n}}) \Rightarrow (G \quad \overline{\Delta}) \vdash ((!\blacksquare)\ \overline{\overline{C}})\ p \downarrow w\ w'\ R\ (0\ \overline{\overline{n}})}\ \downarrow\textsc{Negação}$$

$$\frac{n \neq 0 \qquad n'' = n - 1}{\begin{array}{c}(G \quad \overline{\Delta}) \vdash ((!\blacksquare)\ \overline{\overline{C}})\ p \uparrow (\overline{\bar{t}})\ (\overline{\overline{t'}}\ t'') \bot (n\ \overline{\overline{n'}}) \Rightarrow \\ (G \quad \overline{\Delta}) \vdash ((!\blacksquare)\ \overline{\overline{C}})\ p \uparrow (t''\ \overline{\bar{t}})\ (\overline{\overline{t'}}) \bot (n''\ \overline{\overline{n'}})\end{array}}\ \textsc{Retroceder Negação}$$

$$\frac{}{(G \quad \overline{\Delta}) \vdash ((!\blacksquare)\ \overline{\overline{C}})\ p \uparrow w\ w' \bot (0\ \overline{\overline{n}}) \Rightarrow (G \quad \overline{\Delta}) \vdash (\overline{\overline{C}})\ !p \uparrow w\ w' \top (\overline{\overline{n}})}\ \uparrow\textsc{Negação}$$

$$\frac{n'' = n_1 + n_2}{\begin{array}{c}(G \quad \overline{\Delta}) \vdash ((!\blacksquare)\ \overline{\overline{C}})\ p \uparrow w\ w' \top (n_1\ n_2\ \overline{\overline{n'}}) \Rightarrow \\ (G \quad \overline{\Delta}) \vdash (\overline{\overline{C}})\ !p \uparrow w\ w' \bot (n''\ \overline{\overline{n'}})\end{array}}\ \uparrow\neg\textsc{Negação}$$

Figura 4.28: Semântica *small-step* de negação.

negação. A regra ↑Negação define que uma negação resulta em sucesso se sua subexpressão falha e a regra ↑ ¬Negação, que falha se sua subexpressão resulta em sucesso.

A Figura 4.29 apresenta as regras de transição que tratam restrições. O contexto de expressão de *parsing* $(?\blacksquare\ e)$ sinaliza que uma restrição está sendo avaliada e armazena

$$\boxed{(G \quad \overline{\Delta}) \vdash \overline{C}\ p\ D\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta'}) \vdash \overline{C'}\ p'\ D'\ w''\ w'''\ R'\ \overline{n'}}$$

$$\frac{}{\begin{array}{c}(G \quad \overline{\Delta}) \vdash (\overline{\overline{C}})\ ?e \downarrow w\ w'\ R\ \overline{n} \Rightarrow \\ (G \quad \overline{\Delta}) \vdash ((?\blacksquare\ e)\ \overline{\overline{C}})\ ?e \square w\ w'\ R\ \overline{n}\end{array}} \downarrow\textsc{Restrição}$$

$$\frac{}{\begin{array}{c}(G \quad \overline{\Delta}) \vdash ((?\blacksquare\ e)\ \overline{\overline{C}})\ ?\mathbf{true} \square w\ w'\ R\ \overline{n} \Rightarrow \\ (G \quad \overline{\Delta}) \vdash (\overline{\overline{C}})\ ?e \uparrow w\ w'\ \top\ \overline{n}\end{array}} \uparrow\textsc{Restrição}$$

$$\frac{}{\begin{array}{c}(G \quad \overline{\Delta}) \vdash ((?\blacksquare\ e)\ \overline{\overline{C}})\ ?\mathbf{false} \square w\ w'\ R\ \overline{n} \Rightarrow \\ (G \quad \overline{\Delta}) \vdash (\overline{\overline{C}})\ ?e \uparrow w\ w'\ \bot\ \overline{n}\end{array}} \uparrow \neg\textsc{Restrição}$$

Figura 4.29: Semântica *small-step* de restrição.

sua subexpressão original. A regra ↓Restrição prepara o terreno para a aplicação de uma restrição, adentrando no estágio de avaliação, no qual sua subexpressão deve ser avaliada para um valor. As regras ↑Restrição e ↑ ¬Restrição definem, respectivamente, que uma restrição resulta em sucesso se sua subexpressão é avaliada para **true** e falha sua sua subexpressão é avaliada para **false**.

$$\boxed{(G \quad \overline{\Delta}) \vdash \overline{C}\ p\ D\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta'}) \vdash \overline{C'}\ p'\ D'\ w''\ w'''\ R'\ \overline{n'}}$$

$$\frac{}{\begin{array}{c}(G \quad (\Delta\ \overline{\overline{\Delta'}})) \vdash (\overline{\overline{C}})\ \vartheta = p \downarrow w\ w'\ R\ \overline{n} \Rightarrow \\ (G \quad (\Delta\ \Delta\ \overline{\overline{\Delta'}})) \vdash ((\vartheta = \blacksquare\ w)\ \overline{\overline{C}})\ p \downarrow w\ w'\ R\ \overline{n}\end{array}} \downarrow\textsc{Bind}$$

$$\frac{w = (\bar{\bar{t}}\ \overline{\bar{t'}}) \qquad w' = (\overline{\bar{t'}})}{\begin{array}{c}(G \quad (\Delta_1\ \Delta_2\ \overline{\overline{\Delta'}}) \vdash ((\vartheta = \blacksquare\ w)\ \overline{\overline{C}})\ p \uparrow w'\ w''\ \top\ \overline{n} \Rightarrow \\ (G \quad (\Delta_1[\vartheta/\bar{\bar{t}}]\ \overline{\overline{\Delta'}}) \vdash (\overline{\overline{C}})\ \vartheta = p \uparrow w'\ w''\ \top\ \overline{n}\end{array}} \uparrow\textsc{Bind}$$

$$\frac{}{\begin{array}{c}(G \quad (\Delta_1\ \Delta_2\ \overline{\overline{\Delta'}})) \vdash ((\vartheta = \blacksquare\ w)\ \overline{\overline{C}})\ p \uparrow w'\ w''\ \bot\ \overline{n} \Rightarrow \\ (G \quad (\Delta_2\ \overline{\overline{\Delta'}})) \vdash (\overline{\overline{C}})\ \vartheta = p \uparrow w'\ w''\ \bot\ \overline{n}\end{array}} \uparrow \neg\textsc{Bind}$$

Figura 4.30: Semântica *small-step* de *bind*.

A Figura 4.30 apresenta as regras de transição que tratam *binds*. O contexto de expressão de *parsing* ($\vartheta = \blacksquare\ w$) indica que um *bind* está em execução. Esse contexto armazena os símbolos restantes da entrada em seu estado anterior à execução do *bind*.

As regras ↑Bind e ↑ ¬Bind definem que um *bind* resulta em sucesso se sua subexpressão resulta em sucesso e falha se sua subexpressão falha.

$$\boxed{(G \quad \overline{\Delta}) \vdash \overline{C}\ p\ D\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta'}) \vdash \overline{C'}\ p'\ D'\ w''\ w'''\ R'\ \overline{n'}}$$

$$\frac{\begin{array}{c} G[N] = \overline{\vartheta' :: \tau}^{a}\ \overline{e'}^{b}\ p \\ p_i = \vartheta'_i \leftarrow e_i, 1 \leq i \leq a \qquad p_{a+1} = \varepsilon \\ p'_j = \vartheta_j \leftarrow e'_j, 1 \leq j \leq b \qquad p'_{b+1} = \varepsilon \\ C = (\bigtriangledown\ p_1\ p_2 ...\ p_{a+1}\ \bigcirc\ p\ \bigtriangleup\ \overline{\overline{p'}}^{b+1}\ \circ\ N\overline{e}^{a}\ \overline{\vartheta}^{b}) \end{array}}{\begin{array}{c} (G \quad (\overline{\overline{\Delta}})) \vdash (\overline{\overline{C'}})\ N\overline{e}^{a}\ \overline{\vartheta}^{b} \downarrow w\ w'\ R\ \overline{n} \Rightarrow \\ (G \quad (() \ \overline{\overline{\Delta}})) \vdash (C\ \overline{\overline{C'}})\ p_1 \downarrow w\ w'\ R\ \overline{n} \end{array}}\ \downarrow\text{Não-terminal}$$

$$\frac{}{\begin{array}{c} (G \quad \overline{\Delta}) \vdash ((\bigtriangledown\ p_1\ p_2\ \overline{\overline{p'}}\ \bigcirc\ p''\ \bigtriangleup\ \overline{\overline{p'''}}\ \circ\ p''''')\ \overline{\overline{C}})\ p_1 \uparrow w\ w'\ \top\ \overline{n} \Rightarrow \\ (G \quad \overline{\Delta}) \vdash ((\bigtriangledown\ p_2\ \overline{\overline{p'}}\ \bigcirc\ p''\ \bigtriangleup\ \overline{\overline{p'''}}\ \circ\ p''''')\ \overline{\overline{C}})\ p_2 \downarrow w\ w'\ \top\ \overline{n} \end{array}}\ \uparrow\text{Não-terminal}\bigtriangledown_1$$

$$\frac{}{\begin{array}{c} (G \quad \overline{\Delta}) \vdash ((\bigtriangledown\ p_1\ \bigcirc\ p\ \bigtriangleup\ \overline{\overline{p'}}\ \circ\ p'')\ \overline{\overline{C}})\ p_1 \uparrow w\ w'\ \top\ \overline{n} \Rightarrow \\ (G \quad \overline{\Delta}) \vdash ((\bigcirc\ p\ \bigtriangleup\ \overline{\overline{p'}}\ \circ\ p'')\ \overline{\overline{C}})\ p \downarrow w\ w'\ \top\ \overline{n} \end{array}}\ \uparrow\text{Não-terminal}\bigtriangledown_2$$

$$\frac{}{\begin{array}{c} (G \quad \overline{\Delta}) \vdash ((\bigcirc\ p\ \bigtriangleup\ p_1\ \overline{\overline{p'}}\ \circ\ p'')\ \overline{\overline{C}})\ p \uparrow w\ w'\ \top\ \overline{n} \Rightarrow \\ (G \quad \overline{\Delta}) \vdash ((\bigtriangleup\ p_1\ \overline{\overline{p'}}\ \circ\ p'')\ \overline{\overline{C}})\ p_1 \downarrow w\ w'\ \top\ \overline{n} \end{array}}\ \uparrow\text{Não-terminal}\bigcirc$$

$$\frac{}{\begin{array}{c} (G \quad (\Delta\ \overline{\overline{\Delta'}})) \vdash ((\bigcirc\ p\ \bigtriangleup\ \overline{\overline{p'}}\ \circ\ p'')\ \overline{\overline{C}})\ p \uparrow w\ w'\ \bot\ \overline{n} \Rightarrow \\ (G \quad (\overline{\overline{\Delta'}})) \vdash (\overline{\overline{C}})\ p'' \uparrow w\ w'\ \bot\ \overline{n} \end{array}}\ \uparrow \neg\text{Não-terminal}\bigcirc$$

$$\frac{}{\begin{array}{c} (G \quad \overline{\Delta}) \vdash ((\bigtriangleup\ p_1\ p_2\ \overline{\overline{p'}}\ \circ\ p)\ \overline{\overline{C}})\ p_1 \uparrow w\ w'\ \top\ \overline{n} \Rightarrow \\ (G \quad \overline{\Delta}) \vdash ((\bigtriangleup\ p_2\ \overline{\overline{p'}}\ \circ\ p)\ \overline{\overline{C}})\ p_2 \downarrow w\ w'\ \top\ \overline{n} \end{array}}\ \uparrow\text{Não-terminal}\bigtriangleup_1$$

$$\frac{}{\begin{array}{c} (G \quad (\Delta\ \overline{\overline{\Delta'}})) \vdash ((\bigtriangleup\ p_1\ \circ\ p)\ \overline{\overline{C}})\ p_1 \uparrow w\ w'\ \top\ \overline{n} \Rightarrow \\ (G \quad (\overline{\overline{\Delta'}})) \vdash (\overline{\overline{C}})\ p \uparrow w\ w'\ \top\ \overline{n} \end{array}}\ \uparrow\text{Não-terminal}\bigtriangleup_2$$

Figura 4.31: Semântica *small-step* de não-terminal.

A Figura 4.31 retrata as regras de transição que tratam chamadas de não-terminais. Uma chamada de não-terminal é uma tripla $N\ \overline{e}^{a}\ \overline{\vartheta}^{b}$ na qual o primeiro componente (N) é o não-terminal chamado, o segundo componente ($\overline{e}^{a}$) é a sequência de expressões passadas como parâmetros (que são associadas aos atributos herdados), e o terceiro e último

componente ($\overline{\vartheta}^b$) é a sequência de atributos sintetizados, que são associados às expressões retornadas pela regra de produção.

A regra ↓NÃO-TERMINAL ilustra um processo fundamental da estratégia desenvolvida para tratar chamadas de não-terminais: converter a chamada de não-terminal em um contexto de expressão de *parsing* ($\bigtriangledown\ \overline{=}{p}\ \bigcirc\ p'\ \bigtriangleup\ \overline{\overline{p''}}\ \circ\ p'''$) em que $\overline{=}{p}$ é uma sequência de *updates* que declaram os atributos herdados, p' é o corpo da regra de produção, $\overline{\overline{p''}}$ é uma sequência de *updates* que instanciam os atributos sintetizados e p''' é a chamada de não-terminal convertida. A regra ↓NÃO-TERMINAL empilha um ambiente vazio no topo da pilha de ambientes, que é o escopo da chamada de não-terminal, e dá início ao seu preenchimento.

O contexto de expressão de *parsing* ($\bigtriangledown\ \overline{=}{p}\ \bigcirc\ p'\ \bigtriangleup\ \overline{\overline{p''}}\ \circ\ p'''$) auxilia a regra ↑NÃO-TERMINAL$\bigtriangledown_1$ a preencher o escopo da chamada de não-terminal. A cada vez que essa regra é aplicada, o primeiro *update* em $\overline{\overline{p''}}$ é executado, instanciando um atributo herdado com a correspondente expressão passada como parâmetro.

A regra ↑NÃO-TERMINAL$\bigtriangledown_2$ finaliza a construção do escopo da chamada de não-terminal e dá início à execução do corpo da regra de produção dentro desse escopo.

O contexto de expressão de *parsing* ($\bigcirc\ p'\ \bigtriangleup\ \overline{\overline{p''}}\ \circ\ p'''$) é consultado pelas regras ↑NÃO-TERMINAL$\bigcirc$ e ↑ ¬NÃO-TERMINAL$\bigcirc$, que determinam o resultado de uma chamada de não-terminal. Se a execução do corpo do não-terminal foi bem-sucedida, a regra ↑NÃO-TERMINAL$\bigcirc$ inicia o processo de instanciar os atributos sintetizados.

O contexto de expressão de *parsing* ($\bigtriangleup\ \overline{\overline{p''}} \circ p'''$) auxilia a regra ↑NÃO-TERMINAL$\bigtriangleup_1$ a instanciar os atributos sintetizados. A cada vez que essa regra é aplicada, o primeiro *update* em $\overline{\overline{p''}}$ é executado, instanciando um atributo sintetizado com a correspondente expressão retornada. Esse processo é concluído pela regra ↑NÃO-TERMINAL$\bigtriangleup_2$, que remove o escopo do não-terminal da pilha de ambientes, sinalizando o fim bem-sucedido de sua execução.

A Figura 4.32 apresenta as regras de transição que tratam *updates*. Essas regras fazem uso de três classes de contextos de expressões de *parsing*: ($\vartheta \leftarrow e\ \bigtriangledown$), ($\vartheta \leftarrow e\ \bigcirc$) e ($\vartheta \leftarrow e\ \bigtriangleup$). Esses contextos desempenham três funções: determinar em qual escopo a expressão operando deve ser avaliada, especificar qual ambiente deve ser modificado pelo

$$\boxed{(G \quad \overline{\Delta}) \vdash \overline{C}\ p\ D\ w\ w'\ R\ \overline{n} \Rightarrow (G \quad \overline{\Delta'}) \vdash \overline{C'}\ p'\ D'\ w''\ w'''\ R'\ \overline{n'}}$$

$$\frac{C = (\bigtriangledown\ \vartheta \leftarrow e\ \overline{\overline{p}}\ \bigcirc\ \overline{\overline{p'}}\ \bigtriangleup\ \overline{\overline{p''}}\ \circ\ p''')}{\begin{array}{l}(G \quad \overline{\Delta}) \vdash (C\ \overline{\overline{C'}})\ \vartheta \leftarrow e \downarrow w\ w'\ R\ \overline{n} \Rightarrow \\ (G \quad \overline{\Delta}) \vdash ((\vartheta \leftarrow e\ \bigtriangledown)\ C\ \overline{\overline{C'}})\ \vartheta \leftarrow e \square w\ w'\ R\ \overline{n}\end{array}}\ \downarrow\textsc{Update}\bigtriangledown$$

$$\frac{C \neq ((\bigtriangledown\ \vartheta \leftarrow e\ \overline{\overline{p}}\ \bigcirc\ \overline{\overline{p'}}\ \bigtriangleup\ \overline{\overline{p''}}\ \circ\ p''')\ \overline{\overline{C'}}) \qquad C \neq ((\bigtriangleup\ \vartheta \leftarrow e\ \overline{\overline{p}}\ \circ\ p')\ \overline{\overline{C'}})}{\begin{array}{l}(G \quad \overline{\Delta}) \vdash (\overline{\overline{C}})\ \vartheta \leftarrow e \downarrow w\ w'\ R\ \overline{n} \Rightarrow \\ (G \quad \overline{\Delta}) \vdash ((\vartheta \leftarrow e\ \bigcirc)\ \overline{\overline{C}})\ \vartheta \leftarrow e \square w\ w'\ R\ \overline{n}\end{array}}\ \downarrow\textsc{Update}\bigcirc$$

$$\frac{C = (\bigtriangleup\ \vartheta \leftarrow e\ \overline{\overline{p}}\ \circ\ p')}{\begin{array}{l}(G \quad \overline{\Delta}) \vdash (C\ \overline{\overline{C'}})\ \vartheta \leftarrow e \downarrow w\ w'\ R\ \overline{n} \Rightarrow \\ (G \quad \overline{\Delta}) \vdash ((\vartheta \leftarrow e\ \bigtriangleup)\ C\ \overline{\overline{C'}})\ \vartheta \leftarrow e \square w\ w'\ R\ \overline{n}\end{array}}\ \downarrow\textsc{Update}\bigtriangleup$$

$$\frac{C = (\vartheta \leftarrow e\ S) \qquad S \in \left\{\bigtriangledown, \bigcirc\right\}}{\begin{array}{l}(G \quad (\Delta\ \overline{\overline{\Delta'}})) \vdash (C\ \overline{\overline{C'}})\ \vartheta \leftarrow v \square w\ w'\ R\ \overline{n} \Rightarrow \\ (G \quad (\Delta[\vartheta/v]\ \overline{\overline{\Delta'}})) \vdash (\overline{\overline{C'}})\ \vartheta \leftarrow e \uparrow w\ w' \top \overline{n}\end{array}}\ \uparrow\textsc{Update}\bigtriangledown\ \&\ \uparrow\textsc{Update}\bigcirc$$

$$\frac{C = (\vartheta \leftarrow e\ \bigtriangleup)}{\begin{array}{l}(G \quad (\Delta_1\ \Delta_2\ \overline{\overline{\Delta'}})) \vdash (C\ \overline{\overline{C'}})\ \vartheta \leftarrow v \square w\ w'\ R\ \overline{n} \Rightarrow \\ (G \quad (\Delta_1\ \Delta_2[\vartheta/v]\ \overline{\overline{\Delta'}})) \vdash (\overline{\overline{C'}})\ \vartheta \leftarrow e \uparrow w\ w' \top \overline{n}\end{array}}\ \uparrow\textsc{Update}\bigtriangleup$$

Figura 4.32: Semântica *small-step* de *update*.

update e armazenar a expressão operando original.

O contexto de expressão de *parsing* ($\vartheta \leftarrow e\ \bigcirc$) define que a expressão operando do *update* $\vartheta \leftarrow e$ deve ser avaliada no ambiente corrente, que então passa a abrigar a associação entre atributo e valor realizada por esse *update*.

O contexto de expressão de *parsing* ($\vartheta \leftarrow e\ \bigtriangledown$) define que a expressão operando deve ser avaliada no ambiente subsequente ao topo da pilha de ambientes, enquanto a associação entre atributo e valor é efetuada no ambiente corrente. Isso se dá porque este contexto representa um *update* que declara um atributo herdado. Nesse contexto, o topo da pilha é o escopo da chamada de não-terminal, que foi efetuada sobre o ambiente subsequente.

O contexto de expressão de *parsing* ($\vartheta \leftarrow e\ \bigtriangledown$) define que a expressão ope-

rando deve ser avaliada no ambiente subsequente ao topo da pilha de ambientes, e que a associação entre atributo e valor deve ser efetuada no ambiente corrente. Isso ocorre porque esse contexto representa um *update* utilizado para declarar um atributo herdado, instanciando-o com a adequada expressão passada por parâmetro.

O contexto de expressão de *parsing* ($\vartheta \leftarrow e \bigtriangledown$) indica que o *update* $\vartheta \leftarrow e$ declara um atributo herdado no escopo da chamada de não-terminal, o qual encontra-se no topo da pilha de ambientes. Nesse caso, a expressão operando deve ser avaliada para um valor no ambiente subsequente ao topo da pilha, que, nesse contexto, é o ambiente no qual a chamada de não-terminal foi executada.

Por fim, no contexto de parsing ($\vartheta \leftarrow e \bigtriangleup$), é especificado que o *update* introduz um atributo sintetizado no ambiente em que a chamada do não-terminal ocorreu. O valor associado a esse atributo resulta da avaliação da expressão retornada dentro do escopo da chamada do não-terminal.

As regras $\downarrow$UPDATE$\bigtriangledown$, $\downarrow$UPDATE$\bigcirc$, $\downarrow$UPDATE$\bigtriangleup$ dão início ao processo de avaliar a expressão operando do *update* para um valor. Ao final desse processo, as regras $\uparrow$UPDATE$\bigtriangledown$ & $\uparrow$UPDATE$\bigcirc$ e $\uparrow$UPDATE$\bigtriangleup$ estendem o ambiente adequado com a associação entre atributo e valor denotada pelo *update*.

4.5 Sistema de tipos

Um sistema de tipos é um método sintático utilizado para verificar automaticamente a ausência de certos comportamentos errôneos em um programa, classificando suas construções de acordo com os tipos de valores que computam (PIERCE, 2002).

Esta seção discorre sobre a formalização de um sistema de tipos para PEGwSA. As regras que constituem o sistema de tipos formalizado foram divididas em duas partições: a primeira (Subseção 4.5.1) trata expressões de atributos, a última (Subseção 4.5.3) utiliza-se da anterior para tratar expressões de *parsing*.

O conceito de ambiente de tipos (Γ) é utilizado em ambas partições. Um ambiente de tipos é uma estrutura de dados que abriga a associação entre variáveis (γ) e tipos (τ). Uma variável, por sua vez, pode ser um atributo (ϑ) ou um não-terminal (N). A sintaxe abstrata de variável e ambiente de tipos é apresentada na Figura 4.33.

$$\Gamma ::= \overline{\gamma :: \tau}$$
$$\gamma ::= \vartheta \mid N$$

Figura 4.33: Sintaxe abstrata de ambiente de tipos.

O sistema de tipos utiliza-se de duas operações sobre ambientes de tipos: consulta ($\Gamma[\![\gamma]\!]$) e extensão ($\Gamma[\gamma_1 :: \tau_1\ \gamma_2 :: \tau_2 ...\ \gamma_n :: \tau_n]$ tal que $n \geq 1$). A notação $\Gamma[\![\gamma]\!]$ denota o tipo associado à variável γ no ambiente de tipos Γ. A notação $\Gamma[\gamma_1 :: \tau_1\ \gamma_2 :: \tau_2 ...\ \gamma_n :: \tau_n]$ denota a associação de cada variável (γ_1, γ_2, ..., γ_n) ao seu respectivo tipo (τ_1, τ_2, ..., τ_n) no ambiente de tipos Γ.

O sistema de tipo formalizado é monomórfico, ou seja, uma construção retém o mesmo tipo concreto ao longo de sua vida útil. Mais especificamente, um sistema de tipos ser monomórfico implica que uma vez que uma variável seja assinalada com um tipo, essa variável somente pode ser associada a valores desse mesmo tipo (JUN; MICHAELSON; TRINDER, 2002).

4.5.1 Sistema de tipos de expressões de atributos

O julgamento de tipos da partição que trata expressões de atributos tem a forma $\Gamma \vdash e :: \tau$, que deve ser interpretada como: a expressão de atributos e quando avaliada sobre o ambiente de tipos Γ é assinalada com o tipo τ.

As regras de tipagem desta partição foram distribuídas por diversas figuras. Para evitar redundância, não serão fornecidas explicações por escrito das regras já cobertas pelas figuras. No entanto, serão destacados os comportamentos errôneos que essas regras buscam prevenir.

A Figuta 4.34 apresenta as regras de tipagem de literais, atributos e construtores de listas e mapas. Apenas duas regras tratam de construtores de listas, são elas: Lista Unária e Lista. Ambas atuam sobre a notação $e : e$, que representa uma lista como a junção de cabeça ($\mathbf{e} : e$) e cauda ($e : \mathbf{e}$). Cabeça é o primeiro elemento da lista, cauda é a sublista que abriga os elementos que sucedem a cabeça. A primeira regra (Lista Unária) trata de listas com somente um elemento, ou melhor, listas cujas caudas sejam listas vazias (**nil**); a segunda (Lista), de listas compostas por mais de um elemento.

$$\boxed{\Gamma \vdash e :: \tau}$$

$$\frac{}{\Gamma \vdash i :: \mathit{Bool}}\ \textsc{Booleano} \qquad \frac{}{\Gamma \vdash i :: \mathit{Integer}}\ \textsc{Inteiro}$$

$$\frac{}{\Gamma \vdash s :: \mathit{String}}\ \textsc{Cadeia de Caracteres}$$

$$\frac{\Gamma[\![\vartheta]\!] = \tau}{\Gamma \vdash \vartheta :: \tau}\ \textsc{Atributo}$$

$$\frac{\Gamma \vdash e :: \tau}{\Gamma \vdash e : \mathbf{nil} :: [\tau]}\ \textsc{Lista Unária}$$

$$\frac{\Gamma \vdash e_1 :: \tau \qquad \Gamma \vdash e_2 :: [\tau]}{\Gamma \vdash e_1 : e_2 :: [\tau]}\ \textsc{Lista}$$

$$\frac{\Gamma \vdash e_i :: \mathit{String}, 1 \leq i \leq n \qquad \Gamma \vdash e'_j :: \tau, 1 \leq j \leq n}{\Gamma \vdash < \overline{e/e'}^{\,n} > :: < \tau >}\ \textsc{Mapa}$$

Figura 4.34: Regras de tipagem de literais, atributos e construtores de listas e mapas.

$$\boxed{x \leftarrow \mathbf{nil}}$$

(a) Exemplo de lista vazia.

$$\boxed{y \leftarrow \mathbf{true} : 1 : \text{“}\mathit{yes}\text{”} : \mathbf{nil} \bullet (\text{‘1’} \bullet y \leftarrow \mathbf{tail}\ y) * \bullet \mathbf{head}\ y > 0?}$$

(b) Exemplo de lista heterogênea.

Figura 4.35: Exemplos de expressões de *parsing* que levam a comportamentos errôneos evitados pelas regras Lista Vazia e Lista.

As regras que tratam de construtores de listas deixam implícito que listas vazias ou heterogêneas (compostas por elementos de tipos distintos) são mal tipadas. A Figura 4.35 apresenta dois exemplos do uso indevido de construtores de listas que levam a comportamento errôneos, os quais são devidamente descartados pelo sistema de tipos.

No exemplo de lista vazia (Figura 4.35a), o tipo assinalado ao atributo x é indefinido, pois a expressão **nil** pode representar uma lista de inteiros, de booleanos, de cadeias de caracteres, etc. Há algumas alternativas para definir o tipo assinalado a uma lista vazia, ou melhor, para tratar listas vazias como bem tipadas, tais como: deduzir o tipo da expressão através de um algoritmo de inferência (e. g., se o atributo x foi previamente assinalado com o tipo lista de inteiros, então a expressão **nil**, neste contexto, também

deve ser assinalada com o tipo lista de inteiros), alterar a sintaxe de PEGwSA, inserindo anotações de tipos explicitas para listas vazias, ou simplesmente introduzir polimorfismo no sistema de tipos, e.g., definir um tipo lista genérico que tolere ser posteriormente instanciado para qualquer tipo lista especifico (JUN; MICHAELSON; TRINDER, 2002). Todavia, como almejamos um sistema de tipos monomórfico, não tencionamos alterar a sintaxe de PEGwSA para introduzir anotações de tipos explicitas para listas vazias e a inferência não abarca todos os possíveis contextos em que a expressão pode ocorrer, optamos por tratar listas vazias (**nil**) como mal tipadas.

O exemplo de lista heterogênea (Figura 4.35b) apresenta uma sequência de três expressões. A primeira expressão é a instanciação do atributo y com uma lista heterogênea, cujo o primeiro elemento é do tipo booleano; o segundo, do tipo inteiro; e o terceiro e último, do tipo cadeia de caracteres. Em seguida, há uma iteração que, enquanto a entrada iniciar com o terminal '1', consome o primeiro terminal da entrada e remove a cabeça da lista associada ao atributo y. Por fim, é realizada uma restrição que resulta em sucesso quando constata que a cabeça da lista associada a y é maior que zero. Tolerando a introdução de polimorfismo, constata-se que quando essa sequência é submetida às entradas "" e "11" produz, respectivamente, as seguintes construções inválidas: (**true** > 0) e ("*yes*" > 0). Essas construções são inválidas porque as expressões relacionais, que englobam maior que, operam sobre elementos do tipo inteiro, e não dos tipo booleano e cadeia de caracteres.

Esse exemplo traz à tona dois problemas que surgem ao aceitar listas heterogênas no sistema de tipos. O primeiro é restringir a verificação de tipos a ocorrer em tempo de execução, tendo em vista que o exemplo apresenta um caso específico em que o tipo dos elementos usados em uma operação relacional dependem diretamente da entrada na qual a sequência exemplo é aplicada. O segundo, menos intuitivo, é ou a introdução de polimorfismo ou uma restrição possivelmente significativa da expressividade de PEGwSA.

Se for observado, por exemplo, o valor associado ao atributo y enquanto a sequência exemplo é aplicada à entrada "11", temos que na primeira, segunda e terceira iterações o valor associado a y é, respectivamente, (**true** : 1 : "*yes*" : **nil**), (1 : "*yes*" : **nil**) e ("*yes*" : **nil**). Esse comportamento implica que o tipo associado ao atributo y

alterou-se ao longo de sua vida útil. No início, o tipo assinalado ao atributo era algo como $[Bool, Integer, String]$; em seguida, alterou-se para $[Integer, String]$; e, por fim, estabilizou-se como $[Integer]$. Essa abordagem mantém a expressividade da formalização atual de PEGwSA, e quiça impulsiona essa expressividade, ao custo de introduzir polimorfismo, característica indesejada no sistema de tipos almejado.

Uma outra alternativa, que não resultaria em comportamentos errôneos ao se deparar com o exemplo de lista heterogênea proposto, seria simplesmente não introduzir polimorfismo. Sendo assim, ao final da primeira iteração, ao tentar alterar o tipo do atributo y de $[Bool, Integer, String]$ para $[Integer, String]$, a sequência exemplo seria descartada pelo sistema de tipos. Todavia, quais seriam os impactos dessa abordagem em expressividade?

O primeiro impacto seria a impossibilidade de operar sobre listas de tamanho dinâmico, pois, mesmo considerando construções homogêneas, listas de tamanhos distintos seriam obrigatoriamente associadas a tipos diferentes (e. g. os tipos $[Integer, Integer]$ e $[Integer, Integer, Integer]$ seriam considerados distintos nesse contexto). O tamanho de uma lista associada a um atributo, então, seria o mesmo desde sua declaração até o fim da vida útil desse atributo.

Outro golpe na expressividade estaria no acesso dinâmico a listas, que seria muito mais complexo de realizar. O exemplo de lista heterogênea fornecido seria rejeitado mesmo que os elementos da lista fossem homogêneos, pois ao atualizar o atributo y com listas cujos tamanhos fossem diferentes, implicaria em também alterar o tipo do atributo.

Com o intuito de não reduzir a expressividade nem introduzir polimorfismo, o sistema de tipos formalizado não aceita listas heterogêneas.

De maneira análoga às listas, mapas vazios ou heterogêneos são rejeitados pelo sistema de tipos. A regra Mapa, que trata de construtores de mapas, indica que as chaves de mapas devem ser do tipo cadeia de caracteres, enquanto seus valores podem ser de um determinado tipo arbitrário. Como as chaves devem ser avaliadas para valores do tipo cadeia de caracteres, o tipo mapa somente compreende o tipo com o qual os valores do mapa são assinalados.

A Figura 4.36 apresenta as regras de tipagem que tratam operações aritméticas,

$$\boxed{\Gamma \vdash e :: \tau}$$

$$\frac{\Gamma \vdash e_1 :: Integer \qquad \Gamma \vdash e_2 :: Integer}{\Gamma \vdash e_1 + e_2 :: Integer}\ \textsc{Adição}$$

$$\frac{\Gamma \vdash e_1 :: Integer \qquad \Gamma \vdash e_2 :: Integer}{\Gamma \vdash e_1 \times e_2 :: Integer}\ \textsc{Multiplicação}$$

$$\frac{\Gamma \vdash e_1 :: Integer \qquad \Gamma \vdash e_2 :: Integer}{\Gamma \vdash e_1 - e_2 :: Integer}\ \textsc{Subtração}$$

$$\frac{\Gamma \vdash e_1 :: Integer \qquad \Gamma \vdash e_2 :: Integer}{\Gamma \vdash e_1 \div e_2 :: Integer}\ \textsc{Divisão}$$

$$\frac{\Gamma \vdash e_1 :: Bool \qquad \Gamma \vdash e_2 :: Bool}{\Gamma \vdash e_1 \wedge e_2 :: Bool}\ \textsc{Conjunção}$$

$$\frac{\Gamma \vdash e_1 :: Bool \qquad \Gamma \vdash e_2 :: Bool}{\Gamma \vdash e_1 \vee e_2 :: Bool}\ \textsc{Disjunção}$$

$$\frac{\Gamma \vdash e :: Bool}{\Gamma \vdash \neg e :: Bool}\ \textsc{Negação Lógica}$$

$$\frac{\Gamma \vdash e_1 :: Bool \qquad \Gamma \vdash e_2 :: Bool}{\Gamma \vdash e_1 == e_2 :: Bool}\ \textsc{Igualdade Boolean}$$

$$\frac{\Gamma \vdash e_1 :: Integer \qquad \Gamma \vdash e_2 :: Integer}{\Gamma \vdash e_1 == e_2 :: Bool}\ \textsc{Igualdade Integer}$$

$$\frac{\Gamma \vdash e_1 :: String \qquad \Gamma \vdash e_2 :: String}{\Gamma \vdash e_1 == e_2 :: Bool}\ \textsc{Igualdade String}$$

$$\frac{\Gamma \vdash e_1 :: Integer \qquad \Gamma \vdash e_2 :: Integer}{\Gamma \vdash e_1 > e_2 :: Bool}\ \textsc{Maior que}$$

Figura 4.36: Regras de tipagem de operações aritméticas, lógicas e relacionais.

lógicas e relacionais. Uma operação aritmética deve ser assinalada com o tipo inteiro, assim como seus operandos. Uma operação lógica é assinalada com o tipo booleano, tipo com o qual seus operandos também devem ser assinalados. Temos duas operações relacionais, maior que e igualdade, cada uma tratada pelo sistema de tipos de maneira particular. Uma operação de maior que é assinalada com o tipo booleano, e seus operan-

dos, com o tipo inteiro; já uma operação de igualdade, que deve ser assinalada com o tipo booleano, exige que ambos seus operandos sejam assinalados com um mesmo tipo dentre três: booleano, inteiro e cadeia de caracteres.

$$\boxed{\Gamma \vdash e :: \tau}$$

$$\frac{\Gamma \vdash e :: [\tau]}{\Gamma \vdash \textbf{head}\ e :: \tau}\ \textsc{Head} \qquad \frac{\Gamma \vdash e :: [\tau]}{\Gamma \vdash \textbf{tail}\ e :: [\tau]}\ \textsc{Tail}$$

$$\frac{\Gamma \vdash e_1 ::< \tau > \qquad \Gamma \vdash e_2 :: \mathit{String}}{\Gamma \vdash \textbf{get}\ e_1\ e_2 :: \tau}\ \textsc{Get}$$

$$\frac{\Gamma \vdash e_1 ::< \tau > \qquad \Gamma \vdash e_2 :: \mathit{String} \qquad \Gamma \vdash e_3 :: \tau}{\Gamma \vdash \textbf{put}\ e_1\ e_2\ e_3 ::< \tau >}\ \textsc{Put}$$

Figura 4.37: Regras de tipagem de manipulações de listas e mapas.

A Figura 4.37 encerra a partição que trata expressões de atributos, apresentando regras que atuam tanto na manipulação de listas quanto de mapas.

4.5.2 Testes

Durante o desenvolvimento da formalização de PEGwSA em PLT Redex, utilizamos algumas das funções fornecidas pela linguagem para testar e analisar o comportamento do formalismo, são elas: `judgment-holds`, `test-equal`, `traces` e `apply-reduction-relation*`.

A função `judgment-holds` foi utilizada para testar tanto o sistema de tipos quanto a semântica operacional *big-step*. Essa função recebe como parâmetros um `<julgamento ou relação>` e um `<modelo de termo>` e verifica se o julgamento se sustenta, produzindo uma lista de termos que instanciam o modelo de termo com cada atribuição que satisfaz as variáveis do padrão contidas no julgamento. Não nos aprofundaremos sobre o que são variáveis do padrão neste trabalho, os leitores podem simplesmente pensar nessas variáveis como as meta-variáveis que compõem a formalização de PEGwSA, tais como `o` (o resultado da aplicação de uma expressão de *parsing*), Δ (um ambiente), Γ (um ambiente de tipos), etc.

A função `test-equal` também foi utilizada para testar a semântica operacional

```
(judgment-holds (parse ()
                       [(S () (n) (! (• 1 (A (1) (n)))))
                        (A (n) ((× 2 n)) 2)]
                       (S () (result))
                       (1 3 3 4 5)
                       o
                       Δ)
                (o Δ))
```

Figura 4.38: Exemplo teste da semântica *big-step* relacionado à negação.

```
(test-equal
  (judgment-holds
    (typesPeg ()
              ()
              (* (/ (• 0 ((← odd #f))) (• 1 ((← odd #t)))))
              Γ)
    Γ)
  '())
```

Figura 4.39: Exemplo teste do sistema de tipos relacionado à repetição.

big-step e o sistema de tipos. Essa função recebe como parâmetros dois termos, `<termo obtido>` e `<termo esperado>`, e verifica se eles são iguais. Se a função constatar que os termos são distintos, o teste resulta em falha e a disparidade entre os termos é ressaltada.

As funções `traces` e `apply-reduction-relation*` foram utilizadas para testar a semântica operacional *small-step*. Ambas recebem como parâmetro uma `<redução>`, a primeira produz uma representação gráfica da redução, a última somente exibe o passo final da redução.

A seguir, são apresentados três exemplos emblemáticos de testes utilizando as funções mencionadas, cada um desses exemplos contribuiu significativamente para a descoberta de uma falha no sistema de tipos.

```
(test-equal
  (judgment-holds
    (typesPeg ()
              ()
              (• (/ (• 1 ((← x 1))) (• 2 ((← x "two"))))
                 ((← x #t)))
              Γ)
    Γ)
  '())
```

Figura 4.40: Exemplo teste do sistema de tipos relacionado à escolha ordenada.

Em suma, somente realizamos testes unitários, descrevendo julgamentos arbitrários e, vez ou outra, seus respectivos resultados esperados. Com base nos resultados desses testes, foi possível identificar falhas na formalização de PEGwSA, especialmente falhas relacionadas ao sistema de tipos. A Figura 4.38, por exemplo, possui uma chamada da função `judgment-holds` que, quando executada, produz uma lista de termos vazias, o que indica a ocorrência de um comportamento errôneo durante a aplicação de uma expressão de *parsing*. Todavia, essa mesma expressão de *parsing* que, como indicado pelo exemplo, pode levar a comportamentos errôneos era aceita pelo sistema de tipos até aquele momento.

Trataremos com mais profundidade das falhas encontradas, assim como das adaptações para corrigi-las, na próxima subseção.

4.5.3 Sistema de tipos de expressões de *parsing*

O sistema de tipos que especificamos para PEGwSA é baseado no sistema de tipos para APEG proposto por Cardoso et al. (2019). Muitas regras de tipagem são análogas, salvo as devidas adaptações de APEG para PEGwSA.

Durante o processo de especificação de PEGwSA em PLT Redex (sobretudo na apuração, por meio do emprego de testes escritos à mão, da partição do sistema de tipos que atua sobre expressões de *parsing*), nos deparamos com uma série de expressões que, apesar de acarretarem em erros e outros comportamentos indesejados, eram aceitas pelo sistema de tipos. A fim de garantir que o sistema de tipos rejeite essas e outras expressões semelhantes, foi necessário modificar as regras de tipagem utilizadas até aquele momento.

Com o intuito de apresentar esse processo de descoberta e correção de falhas no sistema de tipos, essa subseção apresenta não só as regras de tipagem que atuam sobre expressões de *parsing*, mas também aborda os comportamentos indesejados que essas regras tencionam evitar, algumas expressões de *parsing* que escapavam ao sistema de tipos e as respectivas adaptações nas regras de tipagem até então insatisfatórias.

O julgamento de tipos da partição que atua sobre expressões de *parsing* tem a forma $(G \quad \Gamma) \vdash p \rightsquigarrow \Gamma'$, que deve ser interpretada como: a expressão de *parsing* p, quando avaliada sobre a PEGwSA G e o ambiente de tipos Γ, produz o ambiente de tipos Γ'.

$$\boxed{(G \quad \Gamma) \vdash p \rightsquigarrow \Gamma'}$$

$$\frac{}{(G \quad \Gamma) \vdash \varepsilon \rightsquigarrow \Gamma} \textsc{Cadeia Vazia}$$

$$\frac{}{(G \quad \Gamma) \vdash t \rightsquigarrow \Gamma} \textsc{Terminal}$$

$$\frac{\Gamma[\![\vartheta]\!] = \mathit{unbound} \qquad \Gamma \vdash e :: \tau}{(G \quad \Gamma) \vdash \vartheta \leftarrow e \rightsquigarrow \Gamma[\vartheta :: \tau]} \textsc{Declare}$$

$$\frac{\Gamma[\![\vartheta]\!] = \tau \qquad \Gamma \vdash e :: \tau}{(G \quad \Gamma) \vdash \vartheta \leftarrow e \rightsquigarrow \Gamma} \textsc{Update}$$

$$\frac{(G \quad \Gamma) \vdash p_1 \rightsquigarrow \Gamma_1 \qquad (G \quad \Gamma_1) \vdash p_2 \rightsquigarrow \Gamma_2}{(G \quad \Gamma) \vdash p_1 \bullet p_2 \rightsquigarrow \Gamma_2} \textsc{Sequência}$$

Figura 4.41: Regras de tipagem que tratam de cadeias vazias, terminais, sequências e *updates*.

A Figura 4.41 apresenta as regras de tipagem que atuam sobre cadeias vazias, terminais, sequências e *updates*. Esses tipos de expressões serão frequentemente utilizadas nos exemplos a seguir.

Cadeias vazias e terminais sempre são aceitos pelo sistema de tipos e não alteram o ambiente de tipos, como definido pelas regras Cadeia Vazia e Terminal.

Duas regras de tipagem atuam sobre *updates*, são elas: Update e Declare. A regra Declare aceita *updates* cujos atributos não estão previamente instanciados. Essa regra assinala ao atributo o tipo da expressão parâmetro. A regra Update, em contrapartida, atua sobre *updates* cujos atributos estão previamente instanciados. Essa regra somente aceita um *updates* se o tipo assinalado ao atributo no ambiente de tipos é o mesmo para o qual a expressão parâmetro é avaliada. As duas regras possuem um único objetivo: impedir a introdução de polimorfismo no sistema de tipos, em outras palavras, impedir que o tipo associado a um atributo qualquer mude ao longo de sua vida útil.

Por fim, a regra Sequência define que as alterações no ambiente de tipos realizadas pela primeira subexpressão de uma sequência são visíveis à subexpressão seguinte.

A Figura 4.42 apresenta um exemplo de verificação de tipos que abrange todas as regras de tipagem retratadas na Figura 4.41.

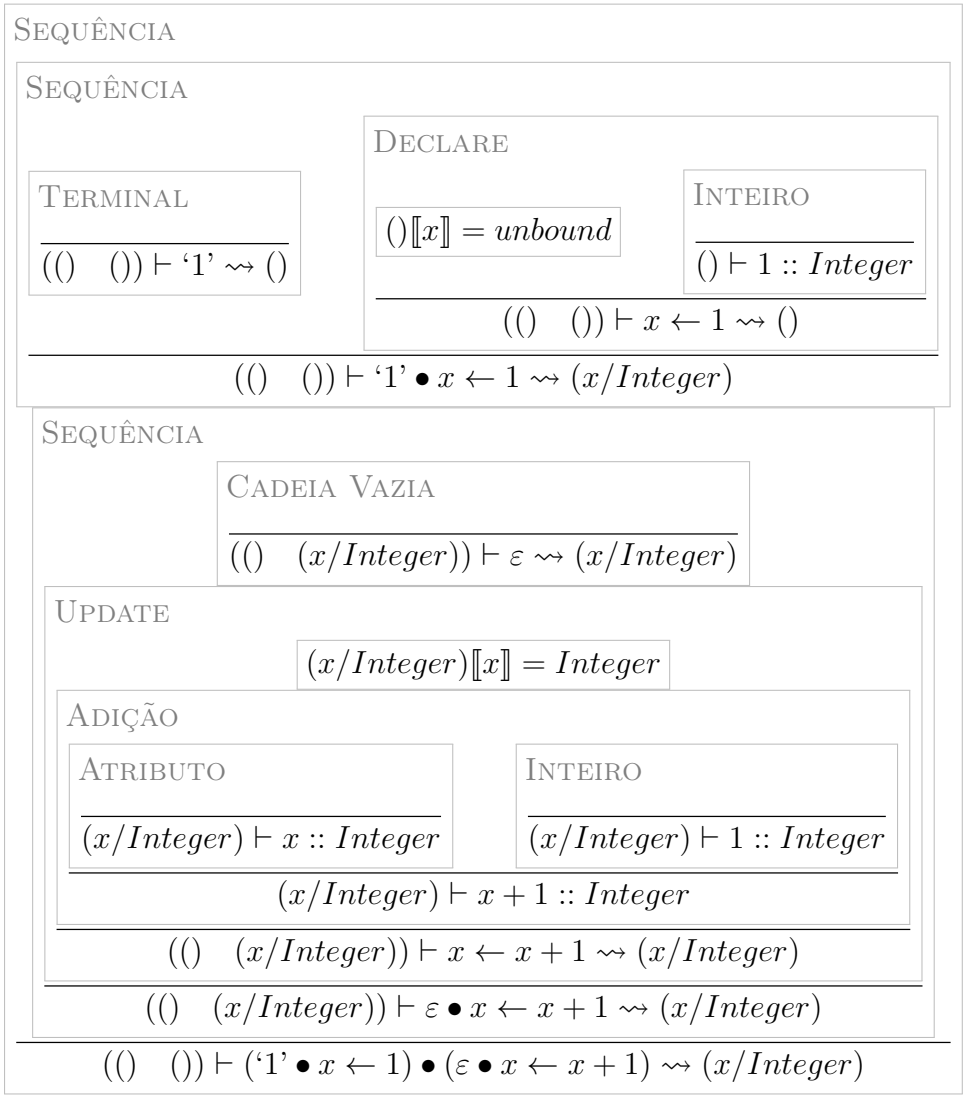

Figura 4.42: Exemplo de verificação de tipos envolvendo todas as regras de tipagem apresentadas na Figura 4.41.

A Figura 4.43 apresenta a regra de tipagem insuficiente que avalia escolhas ordenadas, as possíveis soluções para corrigir essa regra e a solução selecionada para substituí-la, que é a regra de tipagem atual. É necessário ponderar sobre dois aspectos ao definir uma regra de tipagem para tratar escolhas ordenadas: quais escolhas ordenadas devem ser filtradas e como o ambiente de tipos resultante deve ser constituído. A regra de tipagem que trata de escolhas ordenadas, seja qual for, precisa evitar os seguintes comportamentos indesejados: introdução de escopo/polimorfismo e referência a atributos potencialmente não instanciados ou cujos tipos são indefinidos.

Só é possível determinar qual dos ramos de uma escolha ordenada qualquer será efetuado em tempo de execução, e cada ramo pode modificar o ambiente de forma particular. Já que a verificação de tipos é estática, o sistema de tipos é incapaz de determinar

$$\boxed{(G \quad \Gamma) \vdash p \rightsquigarrow \Gamma'}$$

$$\frac{(G \quad \Gamma) \vdash p_1 \rightsquigarrow \Gamma_1 \qquad (G \quad \Gamma) \vdash p_2 \rightsquigarrow \Gamma_2 \qquad \Gamma_1 = \Gamma_2}{(G \quad \Gamma) \vdash p_1/p_2 \rightsquigarrow \Gamma_1} \textsc{Escolha Ordenada}$$

(a) Regra de tipagem atual.

$$\boxed{(G \quad \Gamma) \vdash p \rightsquigarrow \Gamma'}$$

$$\frac{(G \quad \Gamma) \vdash p_1 \rightsquigarrow \Gamma_1 \qquad (G \quad \Gamma) \vdash p_2 \rightsquigarrow \Gamma_2 \qquad \Gamma_1 = \Gamma_2}{(G \quad \Gamma) \vdash p_1/p_2 \rightsquigarrow \Gamma_1} 1^{\text{a}} \textsc{Solução}$$

$$\frac{(G \quad \Gamma) \vdash p_1 \rightsquigarrow \Gamma' \qquad (G \quad \Gamma) \vdash p_2 \rightsquigarrow \Gamma'}{(G \quad \Gamma) \vdash p_1/p_2 \rightsquigarrow \Gamma'} 2^{\text{a}} \textsc{Solução}$$

$$\frac{(G \quad \Gamma) \vdash p_1 \rightsquigarrow \Gamma \qquad (G \quad \Gamma) \vdash p_2 \rightsquigarrow \Gamma}{(G \quad \Gamma) \vdash p_1/p_2 \rightsquigarrow \Gamma} 3^{\text{a}} \textsc{Solução}$$

(b) Possíveis soluções.

$$\boxed{(G \quad \Gamma) \vdash p \rightsquigarrow \Gamma'}$$

$$\frac{(G \quad \Gamma) \vdash p_1 \rightsquigarrow \Gamma_1 \qquad (G \quad \Gamma) \vdash p_2 \rightsquigarrow \Gamma_2}{(G \quad \Gamma) \vdash p_1/p_2 \rightsquigarrow \Gamma_1 \cap \Gamma_2} \textsc{Escolha Ordenada Obsoleta}$$

(c) Regra de tipagem obsoleta.

Figura 4.43: Adaptações em relação a regra de tipagem que avalia escolhas ordenadas.

com precisão o ambiente de tipos produzido por uma escolha ordenada qualquer.

$$\boxed{(\text{`1'} \bullet x \leftarrow 1 \;/\; \text{`2'} \bullet x \leftarrow \text{“}two\text{”}) \bullet y \leftarrow x + 1}$$

(a) Exemplo de referência a atributo cujo tipo é indefinido.

$$\boxed{(\text{`1'} \bullet x \leftarrow 1 \;/\; \text{`2'}) \bullet y \leftarrow x + 1}$$

(b) Exemplo de referência a atributo potencialmente não instanciado.

$$\boxed{(\text{`1'} \bullet x \leftarrow 1 \;/\; x \leftarrow \text{“}two\text{”}) \bullet x \leftarrow \textbf{true}}$$

(c) Exemplo de equivocada introdução de escopo.

Figura 4.44: Exemplos de expressões de *parsing* que levam a comportamentos errôneos evitados pelas regra Escolha Ordenada.

A Figura 4.44 apresenta três expressões de *parsing* que levam a comportamentos

indesejados descartadas pelo sistema de tipos devido à regra Escolha Ordenada. A Expressão 4.44a retrata uma referência a atributo cujo tipo é indefinido. Um atributo cujo tipo é indefinido, nesse caso, é um atributo declarado em ambos os ramos de uma escolha ordenada, mas assinalado com tipos distintos em cada ramo. Uma escolha ordenada pode produzir um ou mais atributos cujos tipos são indefinidos e ainda assim não incorrer em nenhum comportamento errôneo, contanto que não haja nenhuma referência aos atributos em questão fora dos ramos nos quais foram declarados.

A Expressão 4.44a consiste de uma sequência formada por uma escolha ordenada e um *update*. O primeiro ramo da escolha ordenada consome o símbolo '1' e declara x como um atributo do tipo inteiro, já o segundo ramo da escolha ordenada consome o símbolo '2' e declara x como um atributo do tipo cadeia de caracteres. O *update* subsequente à escolha ordenada declara o atributo y, instanciando-o com o valor associado ao atributo x acrescido de um.

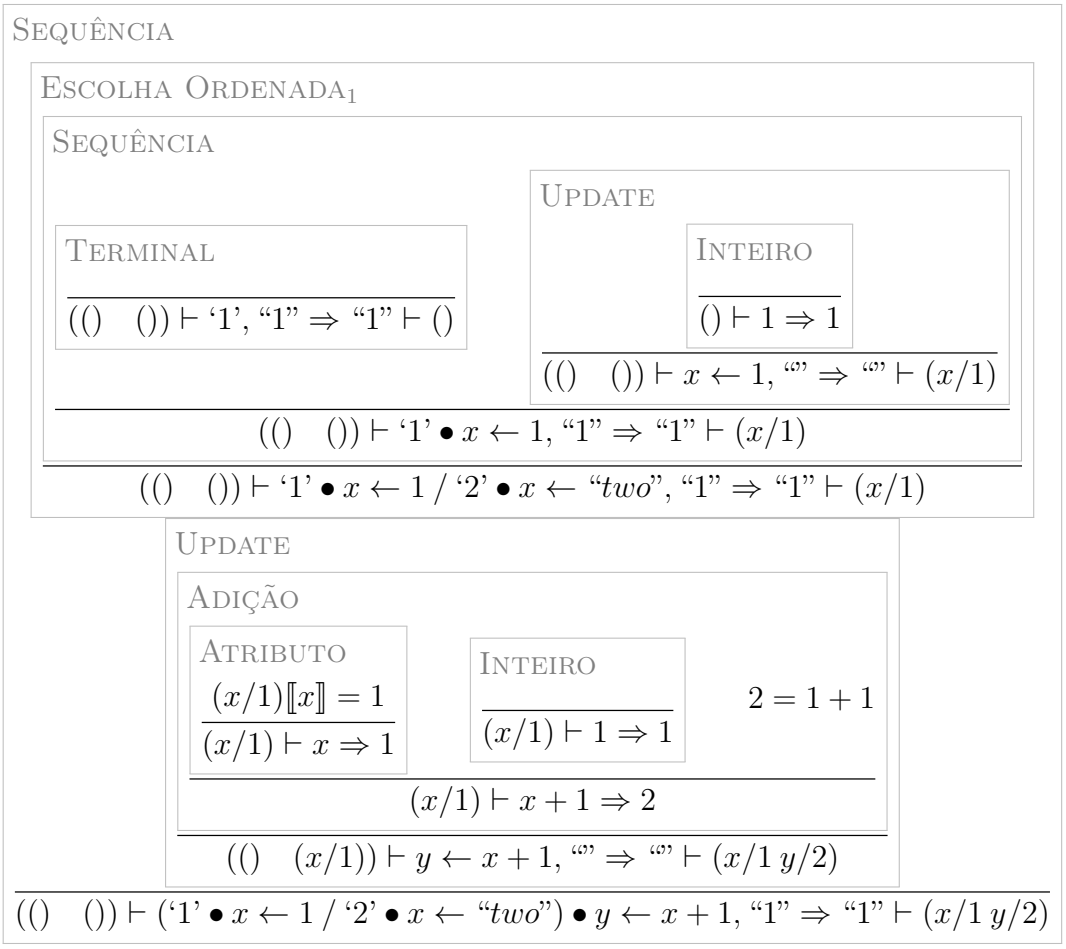

Figura 4.45: Expressão 4.44a aplicada à entrada "1".

Quando aplicada a entradas iniciadas com o símbolo '1', a Expressão 4.44a não

acarreta em nenhum comportamento indesejado e produz um ambiente em que ambos os atributos x e y são associados com valores do tipo inteiro. A Figura 4.45 apresenta a árvore de derivação de um caso particular em que a Expressão 4.44a é aplicada à entrada "1".

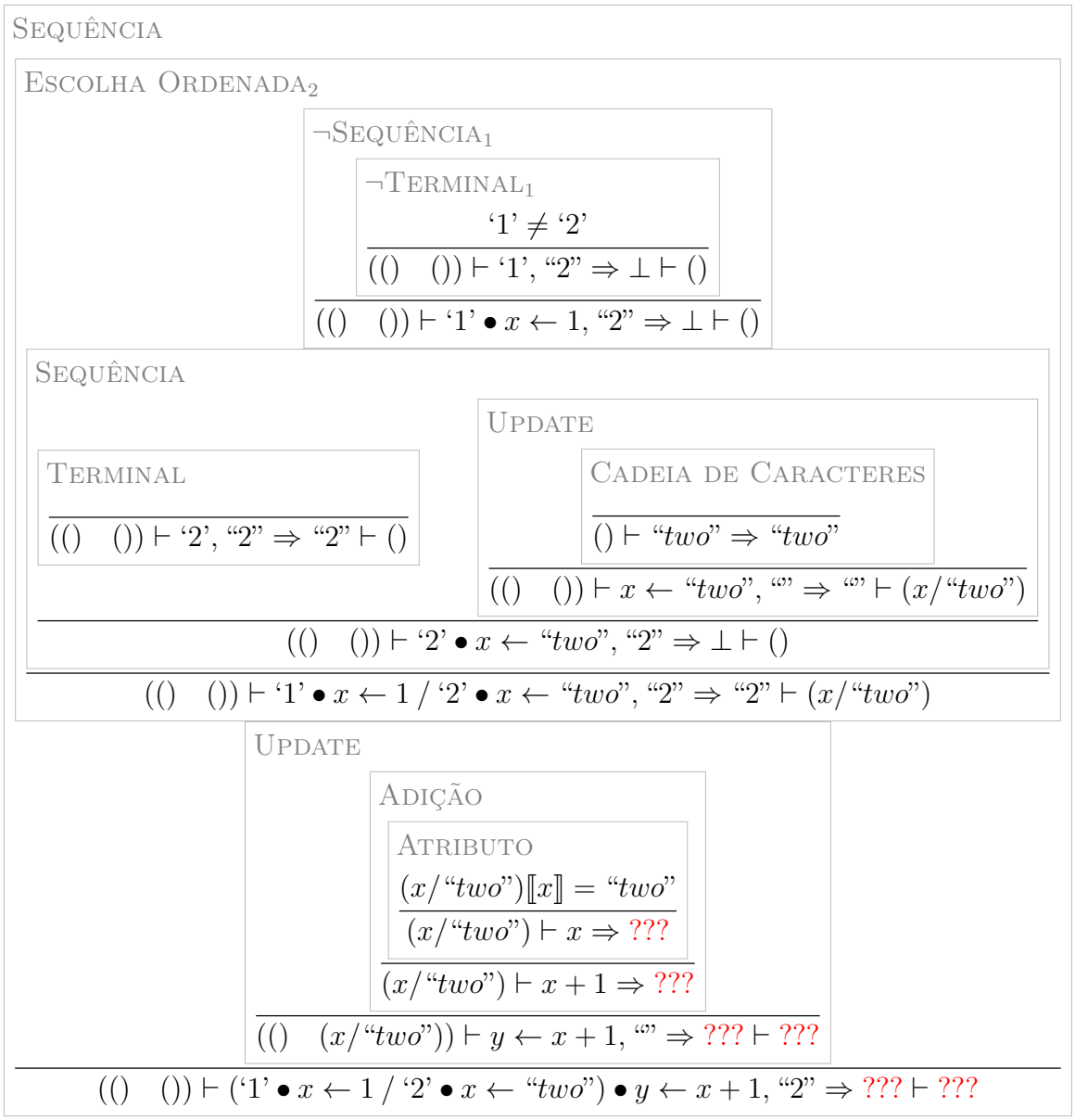

Figura 4.46: Expressão 4.44a aplicada à entrada "2".

De outra forma, se aplicada a entradas iniciadas com o símbolo '2', a Expressão 4.44a incorre em um comportamento errôneo, tendo em vista que o atributo x, que foi associado a um valor do tipo cadeia de caracteres, é utilizado como operando de uma adição, mas a operação de adição é indefinida sobre o tipo cadeia de caracteres, resultando em um estado de *stuck*.

"Estado de *stuck*" é o termo usado para denominar a situação em que é impossível

avaliar uma construção, pois nenhuma regra pode ser aplicada a ela. Definimos o termo "pseudo árvore de derivação" para referenciar a verificação de disparidade entre uma construção e uma regra. Pseudo árvores de derivação objetivam evidenciar os comportamentos errôneos nos quais construções que incorrem em estado de *stuck* geralmente acarretam. A Figura 4.46 apresenta uma pseudo árvore de derivação de um caso particular em que a Expressão 4.44a é aplicada à entrada "2".

As aplicações apresentadas nas Figuras 4.45 e 4.46 demonstram que o tipo que o atributo x assume está diretamente sujeito à entrada sobre a qual a Expressão 4.44a é aplicada, ou seja, que x é um atributo cujo tipo é indefinido. Visto que x, um atributo cujo tipo é indefinido, é referenciado fora da escolha ordenada, a Expressão 4.44a pode ocasionalmente acarretar em comportamentos errôneos (como apresentado na Figura 4.46). Independentemente de qual das regras de tipagem apresentadas na Figura 4.43 seja implementada, o sistema de tipos descarta construções que contém referências a atributos cujos tipos são indefinidos e, consequentemente, evita os comportamentos errôneos que essas referências podem acarretar.

A Expressão 4.44b retrata uma referência a um atributo potencialmente não instanciado, ou seja, uma referência a um atributo que, a depender da entrada sobre a qual a Expressão 4.44b é aplicada, pode ou não ser declarado.

A Expressão 4.44b consiste de uma sequência composta por uma escolha ordenada e um *update*. O primeiro ramo da escolha ordenada consome o símbolo '1' e declara x como um atributo do tipo inteiro, já o segundo ramo da escolha ordenada somente consome o símbolo '2'. O *update* subsequente a escolha ordenada declara o atributo y, instanciando-o com o valor associado ao atributo x acrescido de um.

Caso aplicada a entradas iniciadas com o símbolo '1', a Expressão 4.44b não acarreta em nenhum comportamento errôneo e produz um ambiente em que os atributos x e y são associados com valores do tipo inteiro. A Figura 4.47 apresenta uma árvore de derivação de um caso particular em que a Expressão 4.44b é aplicada à entrada "1".

Por outro lado, se aplicada a entradas iniciadas com o símbolo '2', a Expressão 4.44b incorre em comportamentos errôneos, tendo em vista que o atributo x, referenciado no *update* subsequente à escolha ordenada, não foi previamente declarado nem,

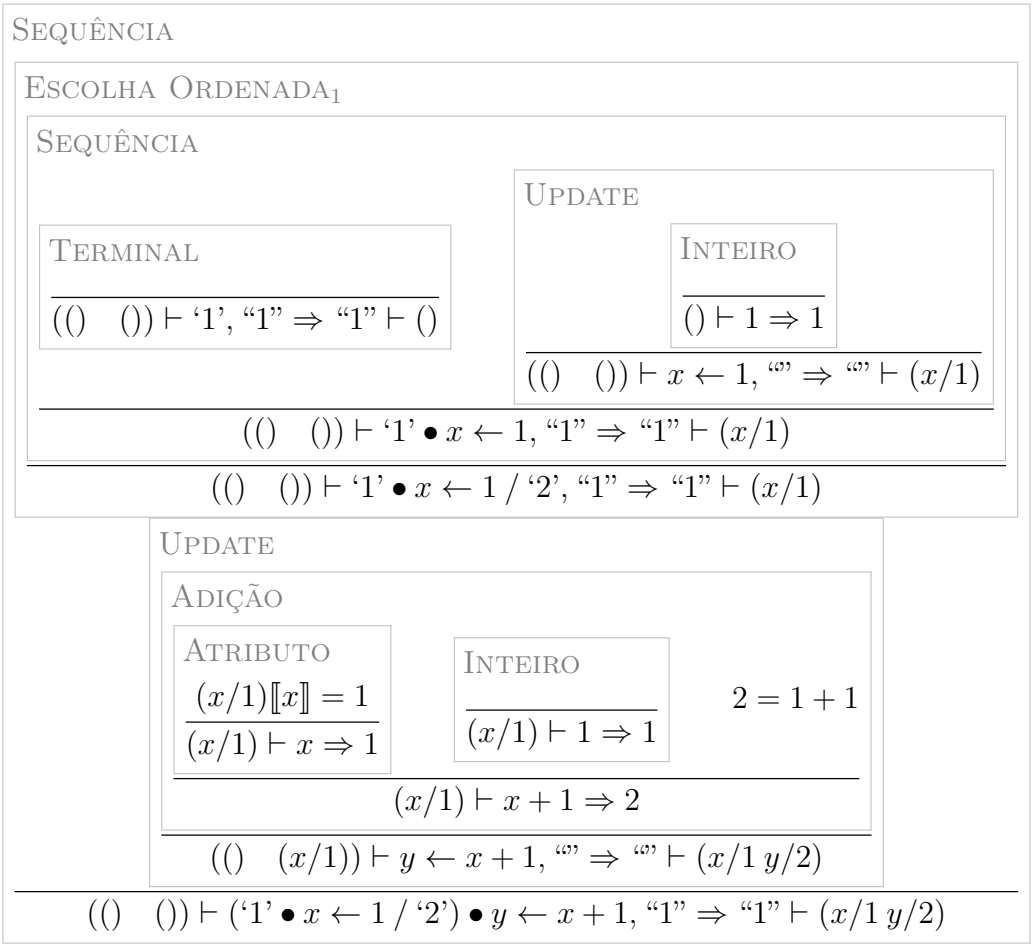

Figura 4.47: Expressão 4.44b aplicada à entrada "1".

em consequência, instanciado. A Figura 4.48 apresenta uma pseudo árvore de derivação de um caso particular em que a Expressão 4.44b é aplicada à entrada "2". Essa pseudo árvore de derivação destaca o exato instante em que o erro de tipagem ocorre: quando o valor associado ao atributo x, até então não declarado, é consultado no ambiente.

Ambos exemplos presentes nas Figuras 4.47 e 4.48 demonstram que, a depender da entrada sobre a qual seja aplicada, a Expressão 4.44b pode incorrer em comportamentos errôneos ou, mais especificamente, referenciar um atributo não instanciado. Independentemente de qual das regras de tipagem presentes na Figura 4.43 seja empregada, o sistema de tipos rejeita construções que contém referências a atributos potencialmente não instanciados.

O terceiro e último exemplo de expressão de *parsing* envolvendo comportamentos errôneos relacionados à tipagem de escolhas ordenadas (Expressão 4.44c) retrata uma construção que, se aceita pelo sistema de tipos, introduz polimorfismo.

A Expressão 4.44c consiste de uma sequência formada por uma escolha ordenada

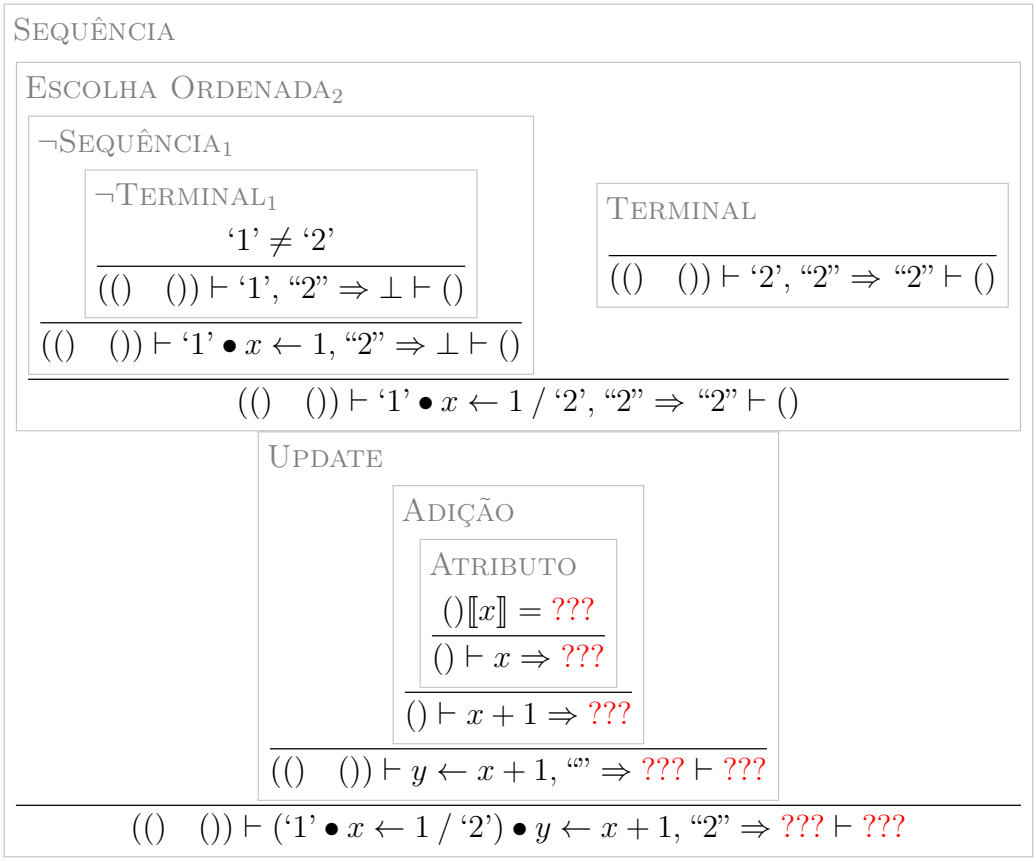

Figura 4.48: Expressão 4.44b aplicada à entrada “2”.

e um *update*. O primeiro ramo da escolha ordenada consome o símbolo ‘1’ e declara x como um atributo do tipo inteiro, já o segundo ramo da escolha ordenada unicamente declara x como um atributo do tipo cadeia de caracteres. O *update* subsequente à escolha ordenada associa ao atributo x um valor do tipo booleano.

Independentemente da entrada sobre a qual a Expressão 4.44c seja aplicada, o tipo assinalado ao atributo x vai ser alterado ao longo da aplicação. A Figura 4.49 apresenta a árvore de derivação de um caso particular em que a Expressão 4.44c é aplicada à entrada “2”. Nessa árvore de derivação, é possível observar que o tipo assinalado ao atributo x mudou de inteiro para booleano ao longo da vida útil do atributo.

A regra de tipagem apresentada na Figura 4.43c é considerada obsoleta porque, se implementada, faz o sistema de tipos aceitar um subconjunto de expressões de *parsing* em que o tipo assinalado a um atributo varia ao longo de sua vida útil, característica indesejada no sistema de tipos almejado.

A diferença entre a regra de tipagem obsoleta e as soluções propostas está na abordagem. A regra de tipagem obsoleta possibilita a rejeição de expressões que levam

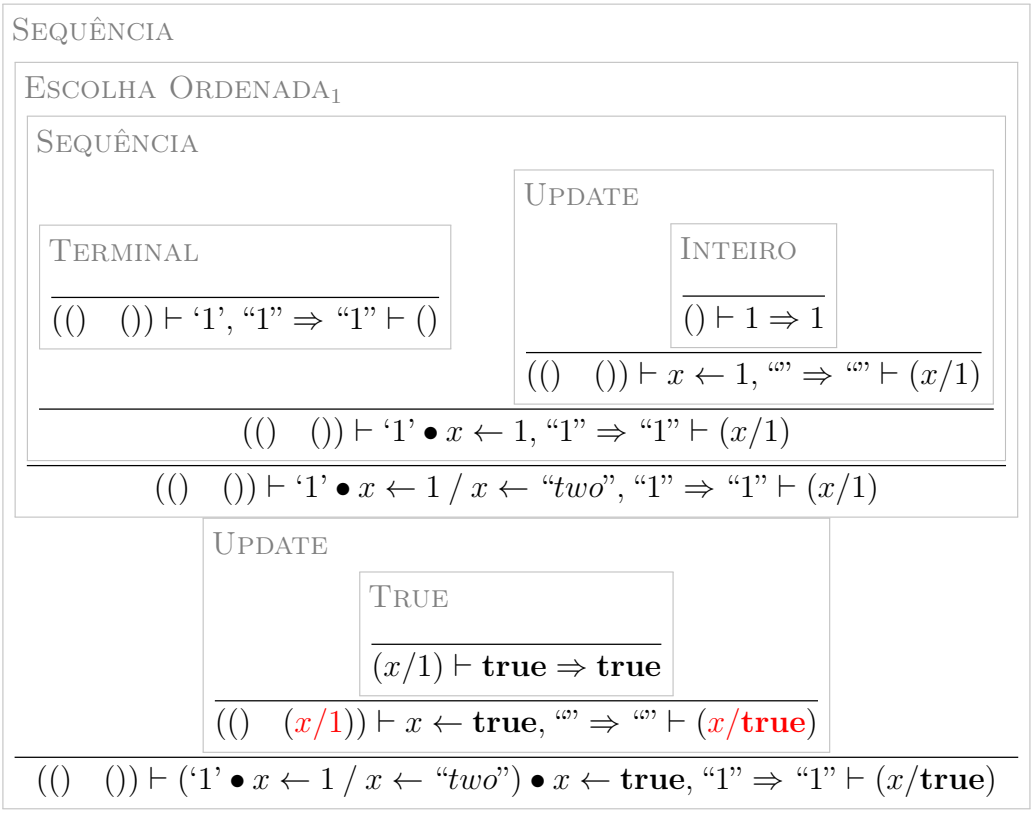

Figura 4.49: Expressão 4.44c aplicada à entrada "1".

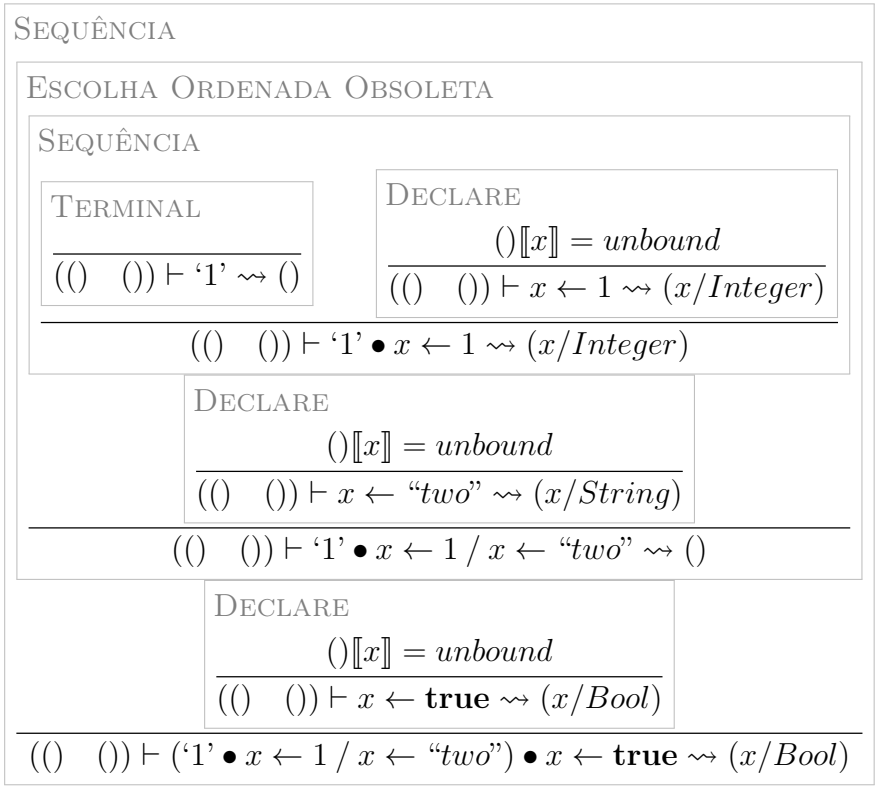

Figura 4.50: Verificação de tipos da expressão de *parsing* retratada na Figura 4.44c ao empregar a regra Escolha Ordenada Obsoleta.

a comportamentos indesejáveis através da composição do ambiente de tipos. Para isso, ela exige que o ambiente produzido por uma escolha ordenada seja a interseção entre os ambientes produzidos por seus ramos. Essa operação de interseção objetiva garantir que o ambiente resultante possua apenas atributos irrevogavelmente declarados e cujos tipos são bem definidos, consequentemente, descartando atributos declarados em somente um ramo ou assinalados com tipos distintos em cada ramo da escolha ordenada. Nesse processo de descartar atributos do ambiente resultante, a regra de tipagem obsoleta torna invisível às expressões fora da escolha ordenada alguns atributos possivelmente declarados dentro dela, ou seja, essa regra cria uma espécie de escopo para cada ramo da escolha ordenada. Em cada um desses escopos, as modificações em atributos declarados fora do ramo sempre são propagas, já os atributos declarados dentro do ramo são propagados se, e somente se, são declarados e assinalados com valores do mesmo tipo em ambos os ramos da escolha ordenada. Uma vez que alguns atributos potencialmente declarados são indetectáveis (visto que não estão presentes no ambiente de tipos), o sistema de tipos torna-se incapaz de prevenir que os tipos assinalados a esses atributos sejam alterados, como retratado na Figura 4.50.

Em contrapartida, as soluções propostas não são voltadas à construção dos ambientes de tipos produzidos por escolhas ordenadas, mas sim à restrição das escolhas ordenadas aceitas pelo sistema de tipos em si. Todas as soluções propostas exigem, em alguma medida, que ambos escolhas ordenadas e seus ramos produzam o mesmo ambiente de tipos. Essa exigência garante que todos os atributos declarados dentro de uma escolha ordenada estão irrevogavelmente declarados, assinalados com tipos bem definidos e visíveis fora da escolha ordenada. A diferença entre as soluções está no quão restritivas elas são.

Se implementada, a terceira e mais restritiva solução leva o sistema de tipos a rejeitar escolhas ordenas que contenham declarações de atributos. As demais soluções, em contrapartida, permitem que atributos sejam declarados dentro de escolhas ordenadas, e por isso são menos restritivas que a terceira.

À primeira vista, é razoável presumir que as duas primeiras soluções aceitam o exato mesmo subconjunto de escolhas ordenadas, mas isso não é verdade, especificamente

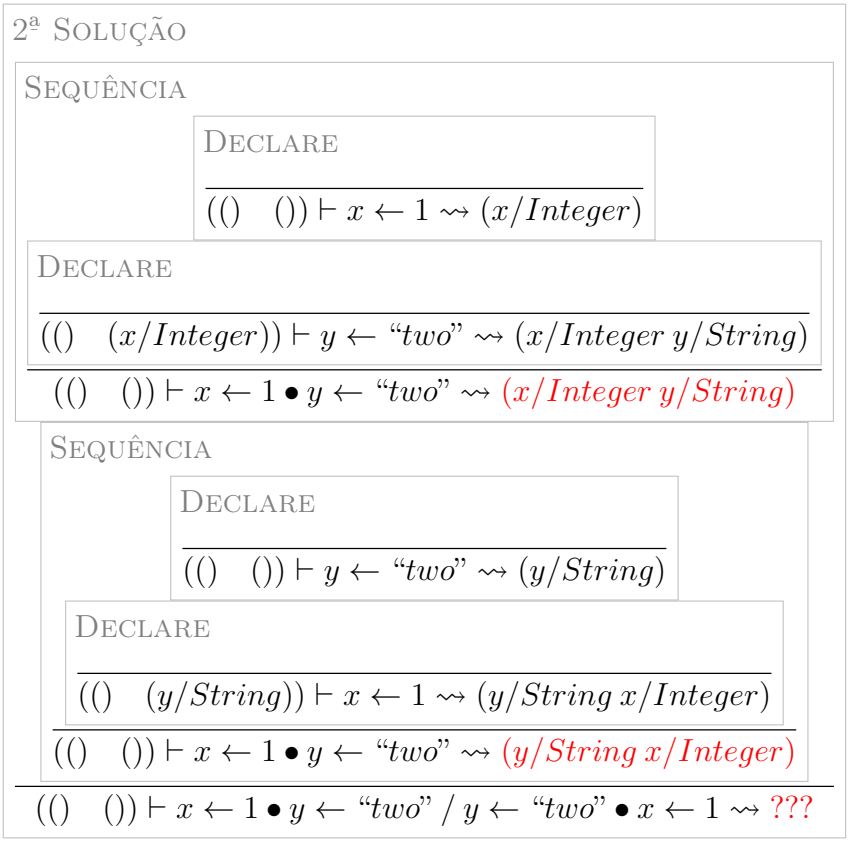

Figura 4.51: Exemplo de expressão de *parsing* rejeitada pelo sistema de tipos se por acaso a segunda solução for implementada.

quando tratamos de PLT Redex. A primeira solução proposta somente exige que ambos os ramos da escolha ordenada produzam o mesmo ambiente de tipos. A segunda solução, por outro lado, é um pouco mais restritiva que a primeira. Na prática, ao tratarmos de PLT Redex, essa solução não somente exige que os ambientes de tipos produzidos por ambos os ramos de uma escolha ordenada sejam iguais, mas também que esses ambientes sejam representados pelo exato mesmo termo. Contudo, um mesmo ambiente pode ser representado por um ou mais termos distintos, e a forma de um termo que representa um ambiente de tipos está sujeita à ordem na qual as declarações de atributos são realizadas. Sendo assim, essa solução rejeita escolhas ordenadas nas quais os mesmos atributos são declarados em ordens diferentes em cada ramo (como demonstrado na Figura 4.51).

Portanto, a primeira solução proposta foi selecionada para substituir a regra de tipagem obsoleta porque, ao permitir a declaração de atributos dentro de escolhas ordenadas, é menos restritiva que a terceira solução e não impõe restrições inusitadas como a segunda.

A Figura 4.52 apresenta adaptações implementados no sistema de tipos com o

$$\boxed{(G \quad \Gamma) \vdash p \rightsquigarrow \Gamma'}$$

$$\frac{\nexists p' : p = !p' \qquad (G \quad \Gamma) \vdash p \rightsquigarrow \Gamma}{(G \quad \Gamma) \vdash !p \rightsquigarrow \Gamma}\ \textsc{Negação}_1 \qquad \frac{(G \quad \Gamma) \vdash p \rightsquigarrow \Gamma'}{(G \quad \Gamma) \vdash !!p \rightsquigarrow \Gamma'}\ \textsc{Negação}_2$$

(a) Regra de tipagem atual.

$$\boxed{(G \quad \Gamma) \vdash p \rightsquigarrow \Gamma'}$$

$$\frac{\nexists p' : p = !p' \qquad (G \quad \Gamma) \vdash p \rightsquigarrow \Gamma}{(G \quad \Gamma) \vdash !p \rightsquigarrow \Gamma}\ 1^{\text{a}}\ \textsc{Solução}_1$$

$$\frac{(G \quad \Gamma) \vdash p \rightsquigarrow \Gamma'}{(G \quad \Gamma) \vdash !!p \rightsquigarrow \Gamma'}\ 1^{\text{a}}\ \textsc{Solução}_2 \qquad \frac{(G \quad \Gamma) \vdash p \rightsquigarrow \Gamma}{(G \quad \Gamma) \vdash !p \rightsquigarrow \Gamma}\ 2^{\text{a}}\ \textsc{Solução}$$

$$\frac{(G \quad \Gamma) \vdash p \rightsquigarrow \Gamma'}{(G \quad \Gamma) \vdash !p \rightsquigarrow \Gamma}\ \textsc{Solução Parcial}$$

(b) Possíveis soluções.

$$\boxed{(G \quad \Gamma) \vdash p \rightsquigarrow \Gamma'}$$

$$\frac{(G \quad \Gamma) \vdash p \rightsquigarrow \Gamma'}{(G \quad \Gamma) \vdash !p \rightsquigarrow \Gamma'}\ \textsc{Negação Obsoleta}$$

(c) Regra de tipagem obsoleta.

Figura 4.52: Adaptações em relação a regra de tipagem que avalia negações.

$$\boxed{!(\text{`1'} \bullet x \leftarrow 1) \bullet y \leftarrow x}$$

(a) Exemplo de referência a atributo potencialmente não instanciado.

$$\boxed{!!(\text{`1'} \bullet x \leftarrow 1) \bullet x \leftarrow \textbf{true}}$$

(b) Exemplo de equivocada introdução de escopo/polimorfismo.

Figura 4.53: Exemplos de expressões de *parsing* que levam a comportamentos errôneos evitados pelas regras $\textsc{Negação}_1$ e $\textsc{Negação}_2$.

intuito de descartar negações que acarretam em comportamentos indesejados. A fim de compreender as motivações por trás dessas adaptações, é pertinente observar o funcionamento do sistema de tipos frente aos exemplos de expressões de *parsing* apresentados na Figura 4.53, assim como o comportamentos das citadas expressões quando aplicadas a determinadas entradas.

A Expressão 4.53a retrata uma expressão de *parsing* que referencia um atributo não instanciado. A expressão em questão consiste de uma sequência composta por uma negação e um *update*. A subexpressão da negação consome o símbolo '1' e declara o atributo x, instanciando-o com um valor do tipo inteiro. O *update* subsequente à negação declara o atributo y, imediatamente instanciando-o com o valor associado ao atributo x.

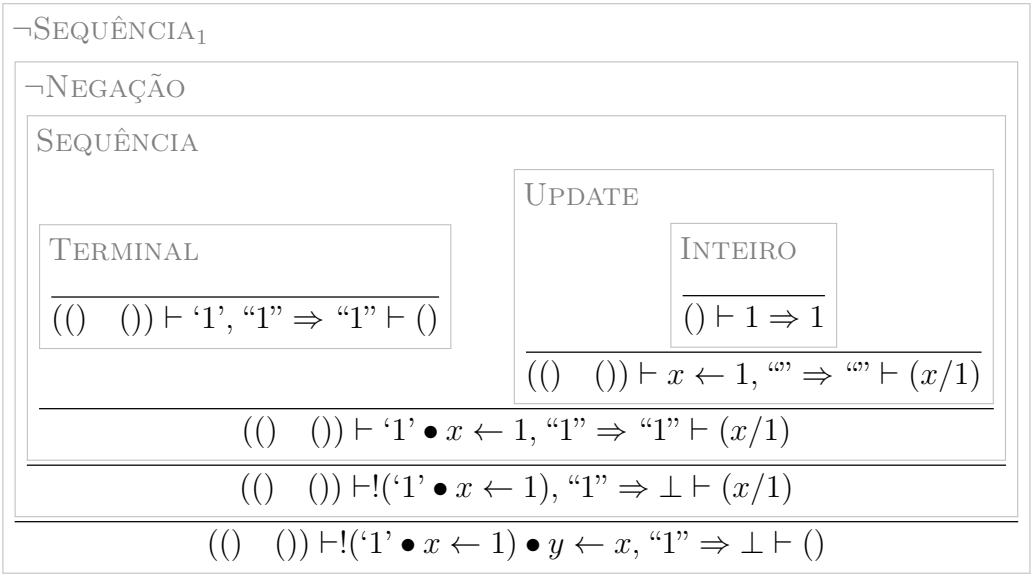

Figura 4.54: Expressão 4.53a aplicada à entrada "1".

Se aplicada a entradas iniciadas com o símbolo '1', a Expressão 4.53a resulta em falha, mas não acarreta em nenhum comportamento errôneo. Um caso particular desse funcionamento pode ser observado na Figura 4.54, em que a árvore de derivação da aplicação da Expressão 4.53a à entrada "1" é retratada.

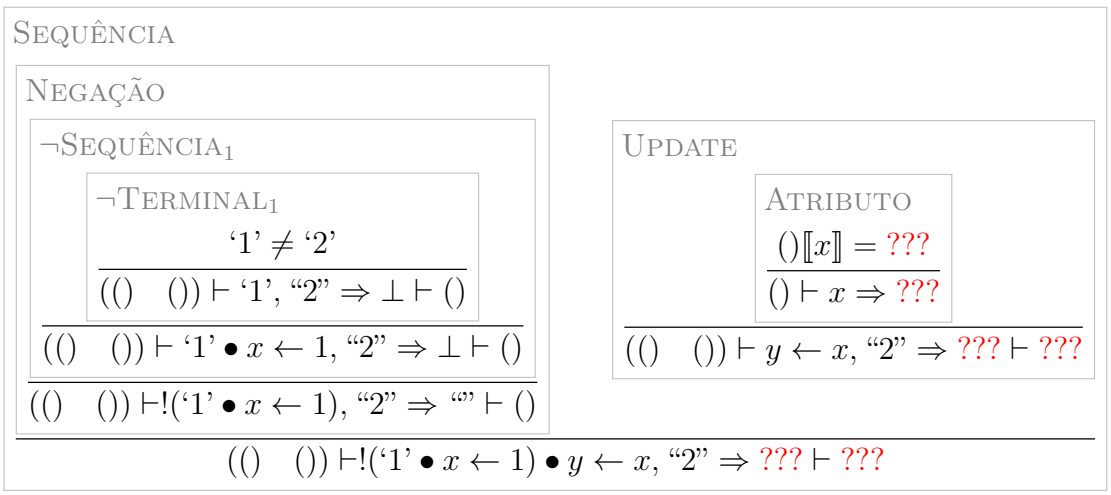

Figura 4.55: Expressão 4.53a aplicada à entrada "2".

Porém, se aplicada a entradas que não iniciem com o símbolo '1', a Expres-

são 4.53a acarreta na referência de um atributo não instanciado. A Figura 4.55 retrata uma pseudo árvore de derivação da aplicação da Expressão 4.53a à entrada "2".

O problema que a Expressão 4.53a visa ressaltar pode ocorrer em expressões que contém referências a atributos declarados dentro de negações. Isso acontece porque, a depender da subexpressão, mudanças no ambiente realizadas dentro de negações podem ou não ser propagadas, o que acarreta em referências a atributos não instanciados.

Se a regra de tipagem obsoleta for implementada, o sistema de tipos aceita a Expressão 4.53a. Por outro lado, se qualquer uma das possíveis soluções propostas (Figura 4.52b) for empregada, o sistema de tipos rejeita a Expressão 4.53a e outras expressões análogas (que contém referências a atributos não instanciados).

A 2ª Solução é a regra de tipagem mais restritiva. Se implementada, o sistema de tipos impossibilita a declaração de atributos dentro de negações. As demais soluções propostas, menos restritivas, permitem a declaração de atributos dentro de negações.

Se a Solução Parcial for implementada, o sistema de tipos rejeita o subconjunto de expressões de *parsing* que contém referências a atributos declarados dentro de negações, evitando assim referências a atributos não instanciados. Todavia, o emprego da solução parcial equivocadamente introduz escopo/polimorfismo no sistema de tipos. Isso ocorre porque a solução parcial descarta quaisquer alterações no ambiente realizadas pelas subexpressões de negações, o que torna os atributos declarados dentro dessas subexpressões indetectáveis ao sistema de tipos, que por sua vez permite que esses atributos sejam novamente declarados e assinalados com tipos quaisquer. Essa característica permite que o tipo assinalado a um atributo seja alterado ao longo de sua vida útil.

A Expressão 4.53b é aceita pelo sistema de tipos se a solução parcial for implementada. A Figura 4.56 retrata uma árvore de derivação da aplicação da Expressão 4.53b à entrada "1". Essa árvore de derivação ressalta que o tipo assinalado ao atributo x foi alterado de inteiro para booleano ao longo de sua vida útil.

A primeira solução, composta por duas regras de tipagem (1ª Solução$_1$ e 1ª Solução$_2$), é , no que diz respeito ao subconjunto de construções aceitas, um meio-termo entre a segunda solução e a solução parcial. Diferente da solução parcial, a primeira solução não apenas permite a declaração de atributos dentro de negações mas também a

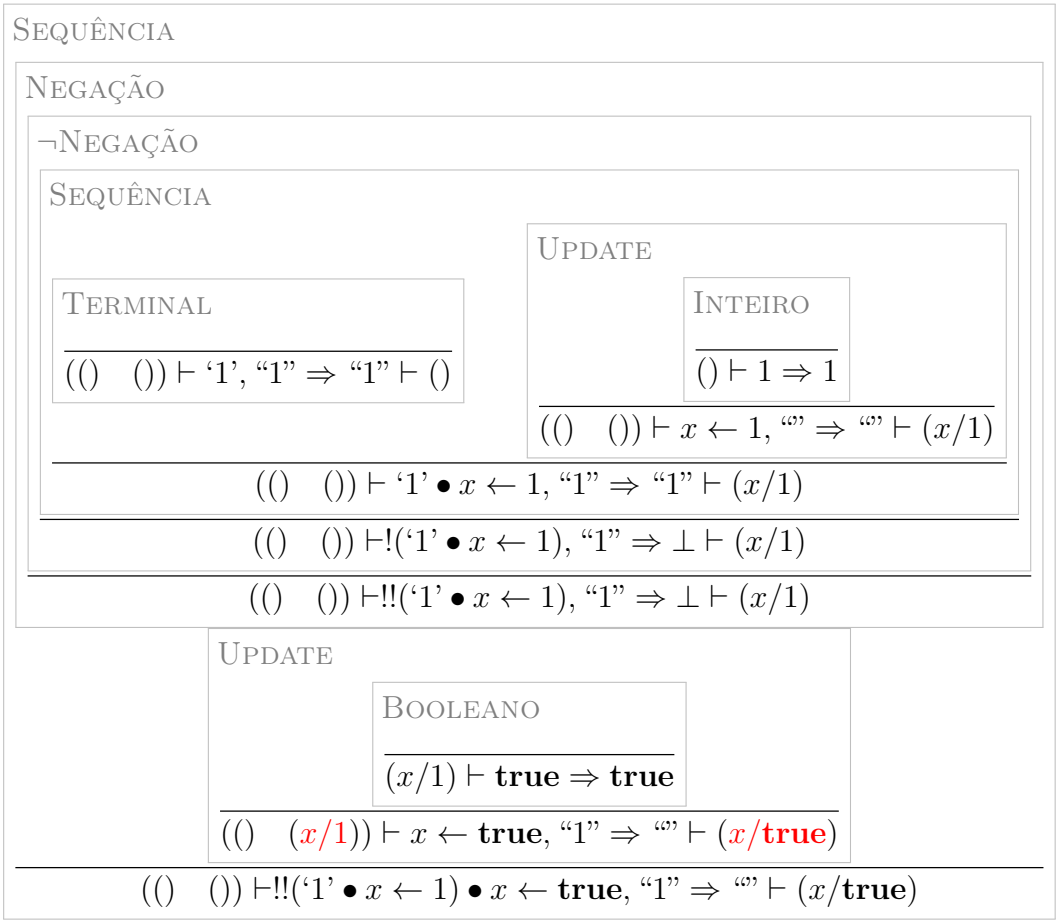

Figura 4.56: Expressão 4.53b aplicada à entrada "1".

propagação desses atributos.

Com o intuito de compreender a abordagem adotada pela primeira solução (Figura 4.52b), é necessário ter em mente a seguinte propriedade semântica de PEGwSA: se, ao resultar em sucesso, a aplicação de uma negação produzir um ambiente diferente do ambiente sobre o qual foi avaliada, então a subexpressão da negação é outra negação

Essa propriedade semântica é alicerçada sobre dois aspectos semânticos de PEGwSA: 1) uma negação resulta em sucesso apenas quando sua subexpressão falha; 2) à exceção da negação, todas os demais tipos de expressões de *parsing* descartam mudanças realizadas no ambiente quando falham.

Além desta, é necessário ter em mente outra propriedade semântica de PEGwSA: uma dupla negação ($!!p$) resulta em sucesso somente se sua subexpressão (p) também resulta em sucesso. Essa propriedade garante a propagação de quaisquer alterações no ambiente realizadas dentro de uma dupla negação.

A primeira solução (Figura 4.52b) tira proveito dessas propriedades ao permitir

que declarações de atributos realizadas dentro de duplas negações sejam propagadas. Por ser menos restritiva que a segunda solução e não introduzir escopo/polimorfismo como a solução parcial, a primeira solução foi selecionada para integrar o sistema de tipos e substituir a regra de tipagem obsoleta.

$$\boxed{(G \quad \Gamma) \vdash p \rightsquigarrow \Gamma'}$$

$$\frac{(G \quad \Gamma) \vdash p \rightsquigarrow \Gamma}{(G \quad \Gamma) \vdash p* \rightsquigarrow \Gamma} \textsc{Repetição}$$

(a) Regra de tipagem atual.

$$\boxed{(G \quad \Gamma) \vdash p \rightsquigarrow \Gamma'}$$

$$\frac{(G \quad \Gamma) \vdash p \rightsquigarrow \Gamma}{(G \quad \Gamma) \vdash p* \rightsquigarrow \Gamma} \text{1ª } \textsc{Solução}$$

$$\frac{(G \quad \Gamma) \vdash p \rightsquigarrow \Gamma'}{(G \quad \Gamma) \vdash p* \rightsquigarrow \Gamma} \textsc{Solução Parcial}$$

(b) Possíveis soluções.

$$\boxed{(G \quad \Gamma) \vdash p \rightsquigarrow \Gamma'}$$

$$\frac{(G \quad \Gamma) \vdash p \rightsquigarrow \Gamma'}{(G \quad \Gamma) \vdash p* \rightsquigarrow \Gamma'} \textsc{Repetição Obsoleta}$$

(c) Regra de tipagem obsoleta.

Figura 4.57: Adaptações em relação a regra de tipagem que avalia repetições.

$$\boxed{(\text{`1'} \bullet x \leftarrow 1) * \bullet y \leftarrow x}$$

(a) Exemplo de referência a atributo potencialmente não instanciado.

$$\boxed{(\text{`1'} \bullet x \leftarrow 1) * \bullet x \leftarrow \textbf{true}}$$

(b) Exemplo de equivocada introdução de escopo/polimorfismo.

Figura 4.58: Exemplos de expressões de *parsing* que levam a comportamentos errôneos evitados pela regra Repetição.

A Figura 4.57 apresenta as adaptações implementadas no sistema de tipos com o objetivo de descartar repetições que acarretam em comportamentos indesejados. Para

isso, são apresentadas a regra de tipagem obsoleta (Figura 4.57c), algumas soluções (Figura 4.57b) e a solução selecionada para integrar o sistema de tipos atual (Figura 4.57a). Para auxiliar na explicação dos comportamentos indesejados que motivam essas adaptações, A Figura 4.58 apresenta dois exemplos de expressões de *parsing* emblemáticos. Os comportamentos errôneos que podem ocorrer com repetições são semelhantes aos que ocorrem com negações.

A Figura 4.58a apresenta um exemplo de expressão de *parsing* que contém uma referência a um atributo potencialmente não instanciado. A Expressão 4.58a consiste de uma sequência composta por uma repetição e um *update*. A subexpressão da repetição consome o símbolo '1' e declara o atributo x, instanciando-o com um valor do tipo inteiro. O *update* subsequente à repetição declara o atributo y, instanciando-o com o valor associado ao atributo x.

Se aplicada a entradas iniciadas com o símbolo '1', a Expressão 4.58a não incorre em nenhum comportamento errôneo, e os atributos x e y são devidamente declarados e instanciados com valores do tipo inteiro. A Figura 4.59 apresenta um caso particular em que a Expressão 4.58a é aplicada à entrada "1".

Em contrapartida, se aplicada a entradas que não são iniciadas com o símbolo '1', a Expressão 4.58a incorre em um comportamento errôneo: a referência a um atributo não instanciado. Esse comportamento indesejado ocorre porque o atributo x referenciado no *update* é declarado dentro da repetição que o procede. Portanto, se a entrada não é iniciada com o símbolo '1', a repetição termina em sucesso sem consumir nenhum símbolo e o atributo x não é declarado nem instanciado. A Figura 4.55 apresenta uma pseudo árvore de derivação de um caso particular em que a Expressão 4.58a é aplicada à entrada "2".

A regra de tipagem obsoleta (Figura 4.57c) aceita expressões de *parsing* que contém declarações de atributos dentro de repetições, ou seja, permite referências a atributos potencialmente não instanciados. Todas as soluções propostas resolvem esse problema.

A solução parcial (Figura 4.57b), apesar de impedir referências a atributos não instanciados, introduz escopo/polimorfismo no sistema de tipos. Isso se dá porque, ao descartar as alterações realizadas no ambiente, essa solução torna os atributos declarados

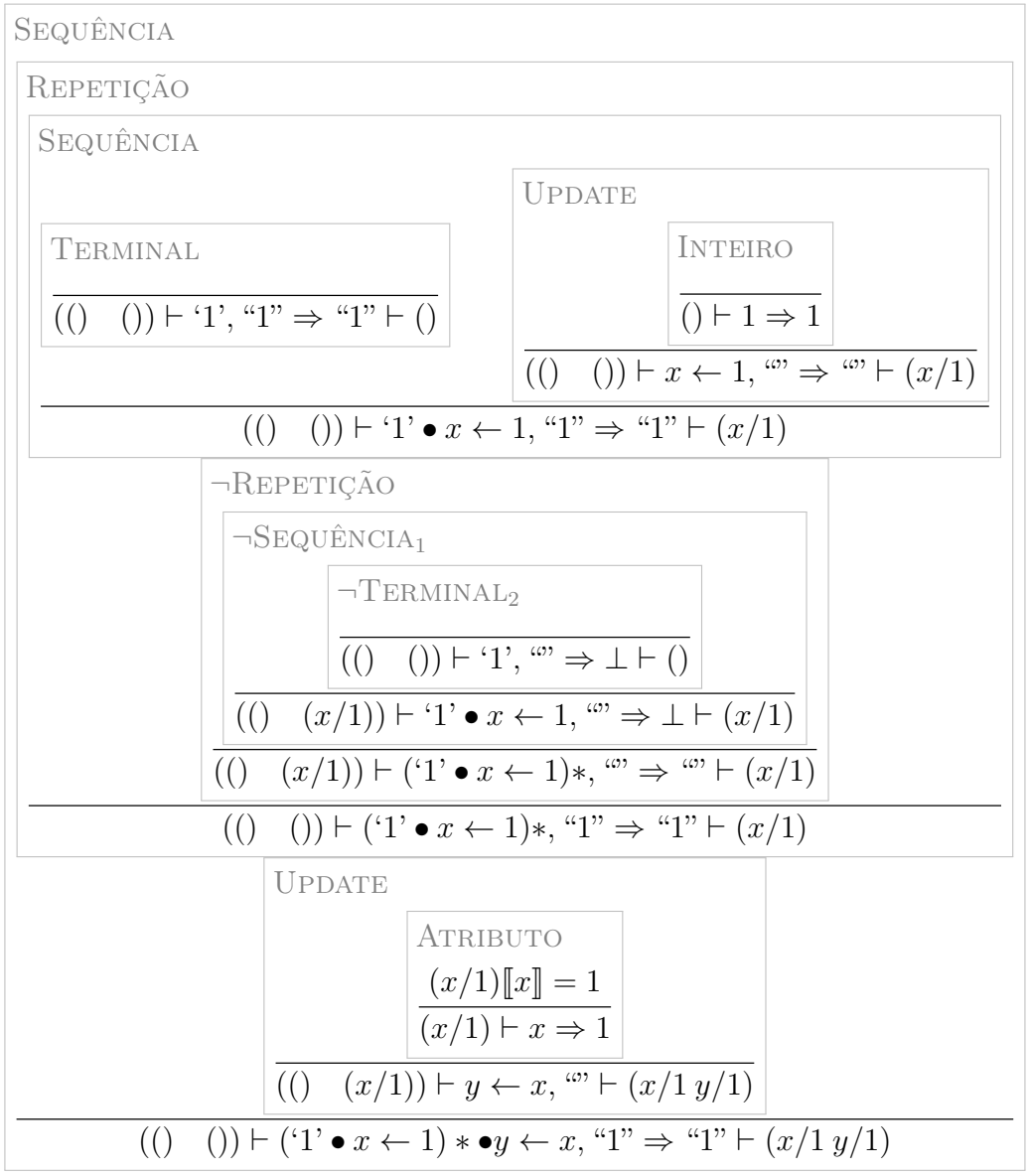

Figura 4.59: Expressão 4.58a aplicada à entrada “1”.

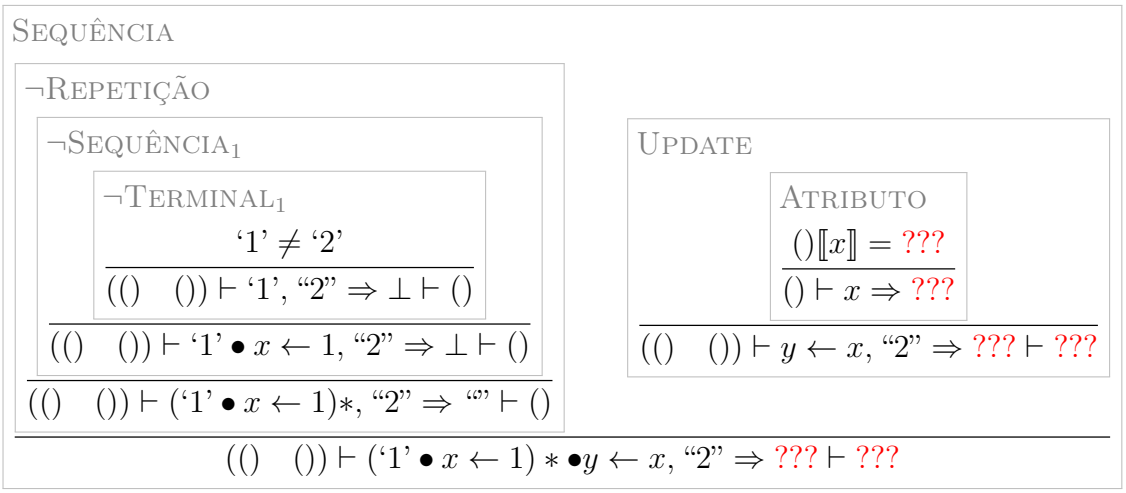

Figura 4.60: Expressão 4.58a aplicada à entrada “2”.

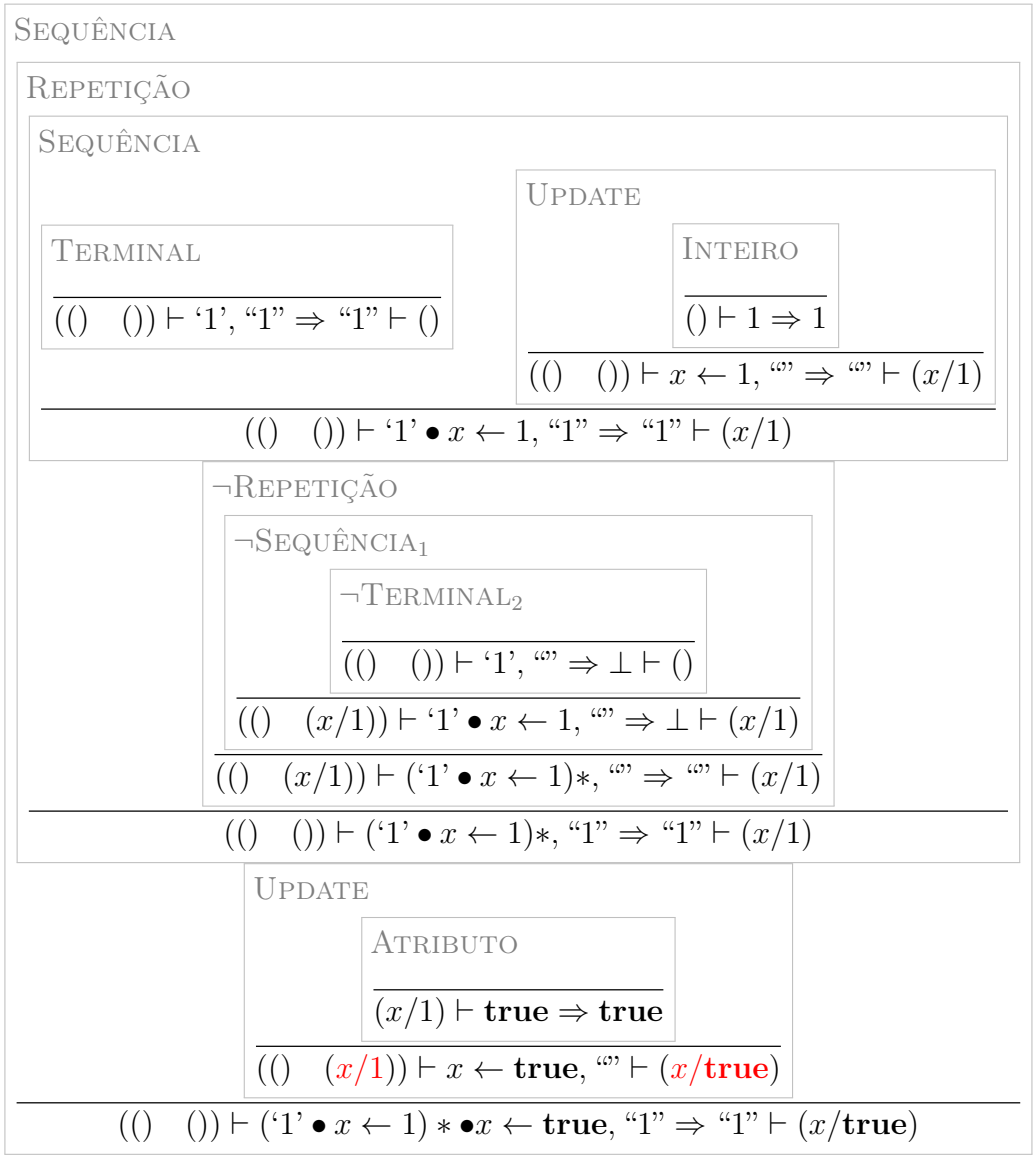

Figura 4.61: Expressão 4.58b aplicada à entrada "1".

dentro de repetições indetectáveis ao sistema de tipos, permitindo que esses mesmos atributos sejam novamente declarados. A Figura 4.53b apresenta um exemplo de expressão de *parsing* que, como demonstrado pela árvore de derivação apresentada na Figura 4.56, possui um atributo cujo tipo pode ser alterado durante sua vida útil.

A primeira solução (Figura 4.57b) é mais restritiva que a solução parcial, pois, ao ser implementada, impede que o sistema de tipos aceite declarações de atributos dentro de repetições. Por impedir referências a atributos potencialmente não instanciados e não introduzir escopo/polimorfismo, a primeira solução foi selecionada para integrar o sistema de tipos e substituir a regra de tipagem obsoleta.

A Figura 4.62 retrata as adaptações realizadas sobre as regras de tipagem que

$$\boxed{(G \quad \Gamma) \vdash p \rightsquigarrow \Gamma'}$$

$$\frac{\Gamma[\![\vartheta]\!] = unbound \qquad (G \quad \Gamma) \vdash p \rightsquigarrow \Gamma' \qquad \Gamma'[\![\vartheta]\!] = unbound}{(G \quad \Gamma) \vdash \vartheta = p \rightsquigarrow \Gamma'[\vartheta :: String]} \text{ Bind Declare}_1$$

$$\frac{\Gamma[\![\vartheta]\!] = unbound \qquad (G \quad \Gamma) \vdash p \rightsquigarrow \Gamma' \qquad \Gamma'[\![\vartheta]\!] = String}{(G \quad \Gamma) \vdash \vartheta = p \rightsquigarrow \Gamma'} \text{ Bind Declare}_2$$

(a) Regras de tipagem atuais.

$$\boxed{(G \quad \Gamma) \vdash p \rightsquigarrow \Gamma'}$$

$$\frac{\Gamma[\![\vartheta]\!] = unbound \qquad (G \quad \Gamma) \vdash p \rightsquigarrow \Gamma'}{(G \quad \Gamma) \vdash \vartheta = p \rightsquigarrow \Gamma'[\vartheta :: String]} \text{ Bind Declare Obsoleta}$$

(b) Regra de tipagem obsoleta.

Figura 4.62: Adaptações em relação a uma das regras de tipagem que avalia *binds*.

atuam sobre *binds* cujos atributos parâmetros não estão previamente instanciados no ambiente de tipos. Como não considera que o atributo parâmetro pode ser declarado dentro de sua subexpressão, a regra de tipagem obsoleta permite a alteração do tipo assinalado a um atributo ao longo de sua vida útil, ou seja, introduz polimorfismo no sistema de tipos.

$$\boxed{x = (\text{`1'} \bullet x \leftarrow 1)}$$

Figura 4.63: Expressão de *parsing* que leva a comportamentos errôneos rejeitada pelo sistema de tipos graças às regras Bind Update, Bind Declare$_1$ e Bind Declare$_2$

A Expressão 4.63 consiste de um *bind* que assinala ao atributo x a porção da entrada consumida por uma sequência que, por sua vez, consome o símbolo '1' e declara o atributo x, instanciando-o com um valor do tipo inteiro. A árvore de derivação da aplicação da Expressão 4.63 à entrada "1" é retratada na Figura 4.64. Nessa árvore de derivação, é destacado uma alteração do tipo de um atributo ao longo de sua vida útil. Visto que almejamos um sistema de tipos monomórfico, a Expressão 4.63 não pode ser aceita pelo sistema de tipos, tendo em vista que essa expressão introduz polimorfismo. Todavia, a regra Bind Declare Obsoleta aceita a Expressão 4.63, e, por isso, é

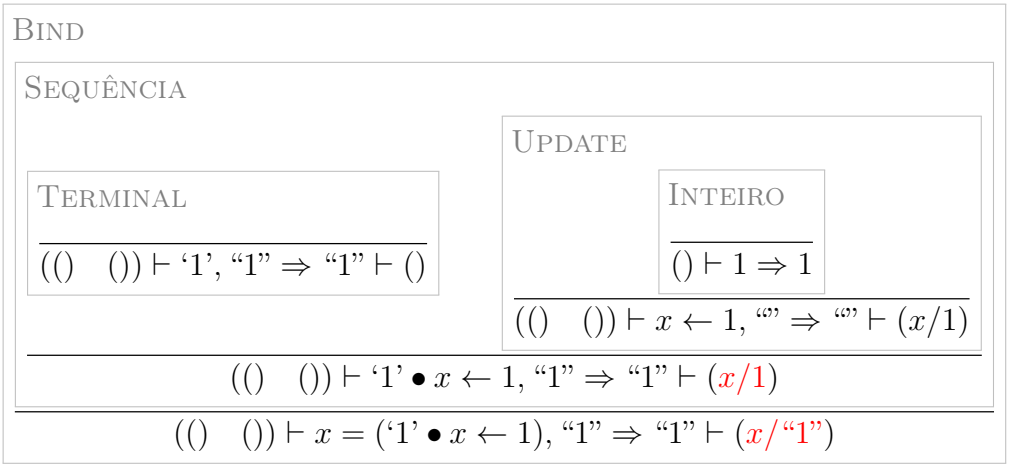

Figura 4.64: Expressão 4.63 aplicada à entrada "1".

insuficiente.

A solução que desenvolvemos é composta por duas regras de tipagem (Bind Declare$_1$ e Bind Declare$_2$). Reunidas, essas regras simplesmente fazem o sistema de tipos rejeitar *bidns* cujos atributos parâmetros são declarados e assinalados em suas subexpressões com tipos diferentes do tipo cadeia de caracteres.

$$\boxed{(G \quad \Gamma) \vdash p \rightsquigarrow \Gamma'}$$

$$\frac{\Gamma[\![\vartheta]\!] = String \qquad (G \quad \Gamma) \vdash p \rightsquigarrow \Gamma'}{(G \quad \Gamma) \vdash \vartheta = p \rightsquigarrow \Gamma'} \text{ Bind Update}$$

$$\frac{\Gamma \vdash e :: Bool}{(G \quad \Gamma) \vdash ?e \rightsquigarrow \Gamma} \text{ Restrição}$$

$$\frac{\begin{array}{c}\Gamma[\![N]\!] = \overline{\tau}^n \to \overline{\tau'}^m \qquad \Gamma \vdash e_i :: \tau_i, 1 \leq i \leq n \\ S = \{\vartheta_j :: \tau'_j | \Gamma[\![\vartheta_j]\!] = unbound, 1 \leq j \leq m\} \\ \forall \vartheta_k \notin S : \Gamma[\![\vartheta_k]\!] = \tau'_k, 1 \leq k \leq m\end{array}}{(G \quad \Gamma) \vdash N\ \overline{e}^n\ \overline{\vartheta}^m \rightsquigarrow \Gamma \cup S} \text{ Chamada de Não-terminal}$$

Figura 4.65: Regras de tipagem que tratam de *binds*, restrições e chamada de não-terminais.

A Figura 4.65 encerra o sistema de tipos, apresentando regras de tipagem que tratam de *binds*, restrições e chamadas de não-terminais. A regra Bind Update atua sobre *binds* cujos atributos parâmetros estão previamente instanciados no ambiente de tipos e permite a propagação dos atributos declarados dentro de suas subexpressões. A regra Restrição rejeita restrições cujas subexpressões não são avaliadas para o tipo

booleano.

Por último, a regra Chamada de Não-terminal atua sobre chamadas de não-terminais. O sistema de tipos assume que todo não-terminal da gramática possui uma anotação de tipos apropriada. A regra Chamada de Não-terminal simplesmente verifica a conformidade entre chamada de não terminal e sua respectiva anotação de tipos.

4.5.4 Algumas observações

O sistema de tipos que formalizamos para PEGwSA não é capaz de capturar certas construções que levam a comportamentos errôneos, tais como aquelas que contém divisão por zero, consulta a lista vazia e busca por uma chave inexistente em um mapa.

Além disso, o sistema de tipos que formalizamos presume que os tipos dos não-terminais já estão devidamente anotados no ambiente de avaliação ao iniciar a verificação de tipos. Como atributos e não-terminais integram um mesmo ambiente de tipos, o sistema de tipos impossibilita a existência de um atributo representado pelo mesmo nome de um não-terminal, equivocadamente considerando essa situação como uma tentativa de alterar o tipo de uma variável ao longo de sua vida útil.

Por fim, o sistema de tipos que definimos aceita gramáticas mal-formadas (FORD, 2004), ou seja, não descarta gramáticas que contém recursões à esquerda nem repetições cujas subexpressões podem terminar em sucesso sem consumir nenhum símbolo.

5 Conclusão

Neste trabalho, apresentamos as especificações em PLT Redex de um sistema de tipos e de semânticas operacionais *big-step* e *small-step* para PEGwSA. Reunidas, essas especificações constituem uma formalização de PEGwSA.

Com o auxílio das ferramentas disponibilizadas pela linguagem PLT Redex, efetuamos uma série de testes sobre essa formalização. Em virtude dos testes mencionados, desenvolvemos um sistema de tipos que agrega aos seus pares — particularmente ao sistema de tipos para APEG apresentado em (CARDOSO et al., 2019) — tendo em vista sua capacidade de capturar construções que provocam comportamentos errôneas que, até então, eram negligenciadas, tais como construções que introduzem polimorfismo ou contém referências a atributos potencialmente não instanciados ou cujos tipos são indefinidos.

Como previamente mencionado, o sistema de tipos para PEGwSA que formalizamos é fortemente embasado no sistema de tipos para APEG apresentado em (CARDOSO et al., 2019) e, em vista disso, herda algumas de suas particularidades, dentre as quais duas são destacadas: 1) aceitar construções mal-formadas; 2) iniciar a verificação de tipos pressupondo que os tipos dos não-terminais foram previamente anotados. Em vista dessas particularidades, podemos propor duas direções para dar continuidade ao desenvolvimento desse trabalho: 1) alterar o sistema de tipos para que ele aceite somente construções bem formadas, como proposto em (FORD, 2004); 2) adaptar para PEGwSA o algoritmo de inferência de tipos para PEG apresentado em (CARDOSO et al., 2023), tendo em vista sua capacidade de identificar os tipos de não-terminais.

Em outra vertente de investigação, é viável utilizar-se dos artefatos desenvolvidos nesse trabalho, sobretudo das especificações em PLT Redex de sistema de tipos e semânticas operacionais, para especificar APEG, formalismo que expande PEGwSA com mecanismos que possibilitam a adaptabilidade dinâmica do conjunto de regras de produção que integram a gramática.

Bibliografia

CARDOSO, E. M. et al. Type-based termination analysis for parsing expression grammars. In: *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*. New York, NY, USA: Association for Computing Machinery, 2023. (SAC '23), p. 1372–1379. ISBN 9781450395175. Disponível em: <https://doi.org/10.1145/3555776.3577620>.

CARDOSO, E. M. et al. An attribute language definition for adaptable parsing expression grammars. In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. New York, NY, USA: Association for Computing Machinery, 2019. (SAC '19), p. 1518–1525. ISBN 9781450359337. Disponível em: <https://doi.org/10.1145/3297280.3299738>.

FILARETTI, D.; MAFFEIS, S. An executable formal semantics of php. In: JONES, R. (Ed.). *ECOOP 2014 – Object-Oriented Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. p. 567–592. ISBN 978-3-662-44202-9.

FORD, B. Parsing expression grammars: a recognition-based syntactic foundation. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 2004. (POPL '04), p. 111–122. ISBN 158113729X. Disponível em: <https://doi.org/10.1145/964001.964011>.

GUHA, A.; SAFTOIU, C.; KRISHNAMURTHI, S. The essence of javascript. In: D'HONDT, T. (Ed.). *ECOOP 2010 – Object-Oriented Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 126–150. ISBN 978-3-642-14107-2.

JUN, Y.; MICHAELSON, G.; TRINDER, P. Explaining polymorphic types. *The Computer Journal*, v. 45, n. 4, p. 436–452, 01 2002. ISSN 0010-4620. Disponível em: <https://doi.org/10.1093/comjnl/45.4.436>.

KLEIN, C. et al. Run your research: on the effectiveness of lightweight mechanization. In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 2012. (POPL '12), p. 285–296. ISBN 9781450310833. Disponível em: <https://doi.org/10.1145/2103656.2103691>.

PIERCE, B. C. *Types and Programming Languages*. 1st. ed. [S.l.]: The MIT Press, 2002. ISBN 0262162091.

POLITZ, J. G. et al. Python: the full monty. *SIGPLAN Not.*, Association for Computing Machinery, New York, NY, USA, v. 48, n. 10, p. 217–232, oct 2013. ISSN 0362-1340. Disponível em: <https://doi.org/10.1145/2544173.2509536>.

ŞERBăNUţă, T.; ROşU, G.; MESEGUER, J. A rewriting logic approach to operational semantics. *Information and Computation*, v. 207, p. 305–340, 02 2009.